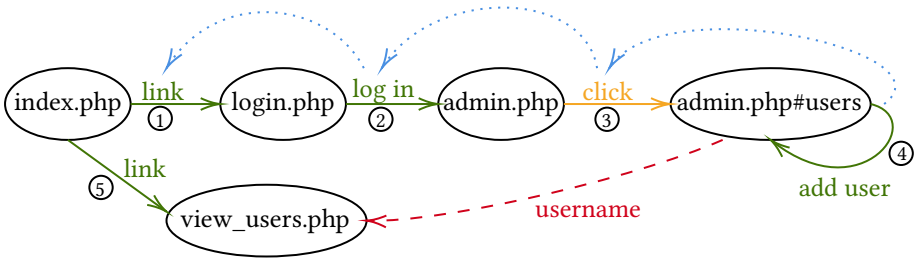
THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Securing the Next Generation Web

Benjamin Eriksson



Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2022

Securing the Next Generation Web

Benjamin Eriksson

ISBN 978-91-7905-680-3
Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 5146
ISSN 0346-718X

Technical report no 220D
Department of Computer Science & Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31-772 1000

Gothenburg, Sweden, 2022

Securing the Next Generation Web
Benjamin Eriksson
Department of Computer Science & Engineering
Chalmers University of Technology

Abstract

With the ever-increasing digitalization of society, the need for secure systems is growing. While some security features, like HTTPS, are popular, securing web applications, and the clients we use to interact with them remains difficult.

To secure web applications we focus on both the client-side and server-side. For the client-side, mainly web browsers, we analyze how new security features might solve a problem but introduce new ones. We show this by performing a systematic analysis of the new Content Security Policy (CSP) directive `navigate-to`. In our research, we find that it does introduce new vulnerabilities, to which we recommend countermeasures. We also create AutoNav, a tool capable of automatically suggesting navigation policies for this directive. Finding server-side vulnerabilities in a black-box setting where there is no access to the source code is challenging. To improve this, we develop novel black-box methods for automatically finding vulnerabilities. We accomplish this by identifying key challenges in web scanning and combining the best of previous methods. Additionally, we leverage SMT solvers to further improve the coverage and vulnerability detection rate of scanners.

In addition to browsers, browser extensions also play an important role in the web ecosystem. These small programs, e.g. AdBlockers and password managers, have powerful APIs and access to sensitive user data like browsing history. By systematically analyzing the extension ecosystem we find new static and dynamic methods for detecting both malicious and vulnerable extensions. In addition, we develop a method for detecting malicious extensions solely based on the meta-data of downloads over time.

We analyze new attack vectors introduced by Google's new vehicle OS, Android Automotive. This is based on Android with the addition of vehicle APIs. Our analysis results in new attacks pertaining to safety, privacy, and availability. Furthermore, we create AutoTame, which is designed to analyze third-party apps for vehicles for the vulnerabilities we found.

Keywords: Web Application Security, Vulnerabilities, Content Security Policy, Web application scanning, Browser extensions, Input validation, Android Automotive.

List of publications

This thesis is based on the following publications, each presented in a separate chapter. Papers A [7], B [5], D [6], F [4] are published at peer-reviewed conferences while Paper C [8] and Paper E [11] are under submission.

Paper A "AutoNav: Evaluation and Automatization of Web Navigation Policies"
Benjamin Eriksson, Andrei Sabelfeld
WWW 2020.

Paper B "Black Widow: Blackbox Data-driven Web Scanning"
Benjamin Eriksson, Giancarlo Pellegrino, Andrei Sabelfeld
S&P 2021.

Paper C "Black Ostrich: Web Application Scanning with String Solvers"
Benjamin Eriksson, Amanda Stjerna, Riccardo De Masellis, Philipp Ruemmer, Andrei Sabelfeld
Manuscript.

Paper D "Hardening the Security Analysis of Browser Extensions"
Benjamin Eriksson, Pablo Picazo-Sanchez, Andrei Sabelfeld
SAC 2022.

Paper E "No Signal Left to Chance: Driving Browser Extension Analysis by Download Patterns"
Pablo Picazo-Sanchez, Benjamin Eriksson, Andrei Sabelfeld
Manuscript.

Paper F "On the Road with Third-Party Apps: Security Analysis of an In-Vehicle App Platform"
Benjamin Eriksson, Jonas Groth, Andrei Sabelfeld
VEHITS 2019.

Acknowledgments

Sometimes it is hard to grasp how many wonderful people have been there for me. Particularly, my supervisor Andrei for being fun, caring, inspiring and awesome! You are always inspiring me and pushing me to try new things, be it traveling across to world for internships, taking on new challenges at work or biking to new cities. A huge thank you to all my amazing colleagues at Chalmers. You all make coming to work, or joining over Zoom, both fun and inspiring! Especially to my PhD buddy Alexander for being there for me since day one, always helping me when I'm lost, whether it is academic, technical, or personal, thank you for being there! Thanks to Iulia for all the fun and insightful discussions, teaching me things outside my bubble. A big thanks to both Mohammad and Ivan for interesting discussions about tradecraft in "Language-Based Security". I also want to thank Christoph from Mozilla for an amazing internship, super fun summer, and great supervision. On the personal side, I owe a lot to Jonas for pushing me in the right direction, giving me the courage to pursue a PhD, and helping me co-author my first paper! I would also like to thank my family for their support and Klas motivating and inspiring my academic pursuits. A special thanks to Agustin, Alejandro, Ann-sofie, Anton and Matti for our amazing trips to Verdansk. Finally, a big thank you to my wife, best friend, and love of my life, Ann-sofie. Thank you for your immense support during this journey, for motivating me to work and for motivating me to do things outside of work.

May 31st, 2022

Contents

Browser Extensions

Embedded Systems

Overview

Introduction

As we are becoming more reliant on digital services the importance of ensuring the services we use are secure is growing. Viewing your medical journals online, banking, and even streaming music and video all rely on secure and efficient software. As more services are being digitalized the related privacy concerns are also growing. But the complex software ecosystems we use, such as the Web, browser extensions, and Android, are difficult to analyze and secure.

Our research efforts can be divided into three parts, the Web, browser extensions, and embedded systems. The Web includes both the client-side, usually a web browser, and the server-side, which is the application on the server. On the client-side the goal is to protect the client from being tricked into executing scripts on the behalf of an attacker in Cross-Site Scripting (XSS) attacks. To mitigate this, modern browsers implement Content Security Policy (CSP). While effective against XSS, it has only recently added a draft for a policy against navigation attacks. Navigation attacks allow an attacker to redirect users from legitimate websites to malicious ones. The problem is that this is a powerful policy that could introduce new unforeseen security problems. Improving server-side security is another challenge that requires us to find security bugs in web applications. Here the big challenge is to develop scanners capable of interacting with web applications and automatically detecting their vulnerabilities. This is challenging as it requires crawling complex workflows in dynamic and state-sensitive applications. In addition, it also requires scanners to submit correct data, including emails, zip codes, or phone numbers, to cover larger portions of web applications.

Browser extensions are small programs that are executed inside the user's web browser. As these have access to powerful APIs, for example for reading users' private data, modifying websites, and monitoring network traffic, it is important to ensure these are not malicious. Furthermore, they can also be vulnerable, opening up for attacks from websites or other extensions. While Google claims to perform vetting of extensions, it is unknown exactly what they do, and sometimes both malicious and vulnerable extensions are missed. Solving the problem of detecting both vulnerable and malicious extensions is an important step to improve the security of browser extensions.

Finally, for embedded systems, we observe that these systems are becoming more complex and many want to support third-party apps. Similar to browser extensions, this requires rigorous vetting before apps should be allowed to run. As Google is

working on their in-car infotainment OS Android Automotive, we expect more cars to support third-party apps.

The goals of this thesis are to:

1. Improve client-side security by analyzing the new navigation policy (Paper A [7]). Design novel methods for automatically analyzing web applications to detect and mitigate web vulnerabilities (Paper B [5] and Paper C [8]).

2. Expand the threat model and develop new methods to analyze and detect both malicious and vulnerable extensions (Paper D [6] and Paper E [11]).

3. Systematically analyze Android Automotive and develop methods to detect malicious apps in order to improve security in embedded systems (Paper F [4]).

I.1 Attackers

The web is a complex ecosystem that allows attackers to use a multitude of different attack vectors. To efficiently protect against these attackers it is crucial to understand their capabilities. The security literature [12] divides the attackers into four classes of attackers, injection, gadget, web, and network attackers. In this thesis we also include the extension attacker.

Injection The injection attacker is the classic web application user. They can interact with the application to perform available actions, for example, comment on images, make their posts, leave reviews, etc. By carefully choosing which actions to perform, the attacker might be able to inject their JavaScript code into the web application. As the attacker is not part of the website, this is considered a Cross-Site Scripting (XSS) attack.

Gadget A gadget is a third-party code that is willingly being included on a website. Common examples are analytic scripts and frameworks, such as jQuery. A gadget attacker is an attacker that can change the gadget code, thus being able to attack multiple websites at the same time. Consider if `mail.com` includes the script `evil.com/analytics.js` then a gadget attacker would try to attack `mail.com` by changing the `analytics.js` code.

Web The unique capability of the web attacker is that they can host their website on the web. The attacker-controlled website can be used to redirect users to malware or force users to send requests to other websites.

The attacker-controlled website `evil.com` can force a user to initiate a request to `mail.com`. If the user is already authenticated with `mail.com` then the attacker could potentially *forge* a request to delete all the user's emails on `mail.com`. This is known as a Cross-site request forgery (CSRF) attack.

Extension Extension attackers have the capability to run code inside browser extensions used by the browser. These small program can use powerful APIs to read privacy-sensitive data like history and cookies, as well as, interact with network traffic. They can also read and modify the content of any webpage the user visits.

For example, if a user visits `mail.com` the extension can read all the email content the user access. Or when using a search engine, the extension can use network APIs to exfiltrate the queries to third-party analytics engines.

Network While the network attacker is out-of-scope for this thesis it is still worth mentioning for a holistic understanding of the ecosystem. The final and strongest attacker is a network attacker. There are two types of network attackers, passive and active. A passive network attacker is capable of listening to all the traffic between the user and the website, while an active network attacker can also modify the traffic.

With this capability, the attacker can record passwords being sent to the website for later account takeovers. In the case of a bank application, an active network attacker would be able to change the recipient bank account of a transaction while it is being sent to the website.

I.2 Web applications

Any website you visit on the web can be considered a web application. When you visit the website your web browser will send a request to the web application, which will be handled by the application code running on the web server. Once handled, the web application will respond with a complete web page.

For a secure web, it is important to improve the security of both the web browsers and web applications, as well as, the interaction between them. The following sections will cover the both the security problems on the client and server-side and our contributions to improving the security.

I.2.1 Client-side

The challenge with client-side security is mainly ensuring that a web attacker is not able to trick the client, commonly a web browser like Chrome or Firefox, to interact with another website in an unsafe manner. We can use the CIA triad to define security as protecting the confidentiality, integrity, and availability of services online. For example, using JavaScript `evil.com` can use `fetch` to read data from other URLs. Here it is important that the browser ensures that the website can not read your emails from another website, such as `mail.com`.

The most notorious client-side attack on the web is XSS. In this attack, `evil.com` can execute JavaScript on `mail.com`. This is accomplished by injecting data that contains HTML, for example: `<script>alert(1)</script>`. While any of the attackers in Section I.1 can launch this attack, the injection attacker and the web attacker are the most common. If the web application on `main.com` is programmed incorrectly, it might output this data as HTML code. In this case, it would result in the JavaScript being executed and a popup being shown to the user. A more malicious attacker could leverage this JavaScript execution to steal passwords and other credentials.

Finally, navigation attacks is a newer type of client-side attack. In a navigation attack, users on `mail.com` could be redirected away to a malicious website like `ev`

`il.com`. This happened on Equifax in 2017 when its users were redirected to sites containing malware [1].

Background. There are different methods to help mitigate these problems. The most fundamental security feature is Same-Origin Policy (SOP). SOP isolates different origins, where an origin is defined as a triplet of protocol (e.g. HTTP), host (e.g. evil.com), and port (e.g. 80). This means that `evil.com` and `mail.com` are different origins and, as such, attempts to fetch data from each other should be blocked by the browser. There are some exceptions to this rule, for example, `evil.com` could load images and scripts from `mail.com`.

However, SOP is not enough to stop XSS as this attack is technically executed inside the same origin. Although it is best to fix the problem in the code on the server-side, as we will explain in Section I.2.2, the browser can also help by using Content Security Policy (CSP) [16]. CSP is a mechanism for websites to define security policies that the browser will enforce. For example, websites can define a policy to only allow loading JavaScript from `mail.com`. Crucial for XSS is that CSP can be used to block all inline JavaScript, which is the main attack vector for XSS.

To mitigate navigation attacks the draft for CSP Level 3 [15] includes a new directive, `navigate-to`. By using this new directive, Equifax could define the following policy to only allow navigations to their domain.

```
navigate-to: equifax.com
```

But while this seems like a nice solution to battle navigation attacks, it might simultaneously introduce new problems.

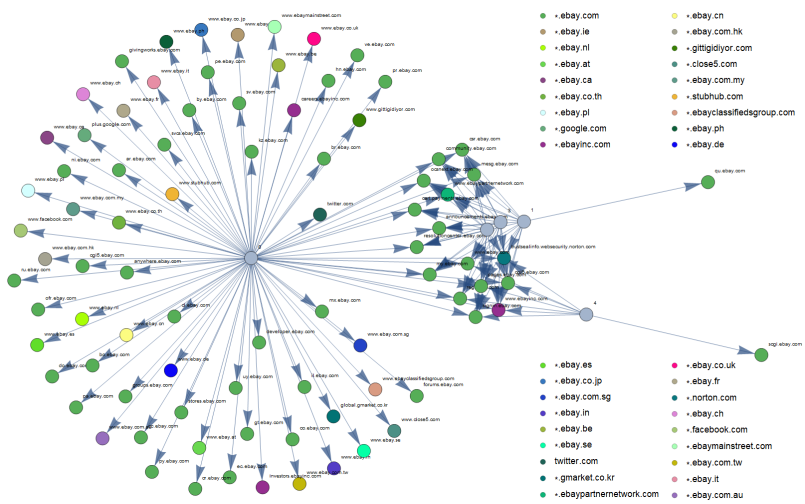


Figure I.1: Generated policies for `ebay.com`. The nodes with outwards pointing arrows are the five pages that we crawled. All the other nodes correspond to a possible navigation. The color indicates which part of the policy covers the navigation.

Contribution. In paper A [7] we research this new navigation policy to test both if it introduces new vulnerabilities and what the performance impact on the web is. We discover that it efficiently protects against navigation attacks but also introduces new methods to probe users for private data. In particular, the web attacker from Section I.1 can abuse this policy to gain information about the user visiting their website. To help the adaptation of this new policy we develop AutoNav in Paper A [7]. AutoNav is an open-source tool developers can use to scan their websites for outgoing navigations, mostly hyperlinks, and then get suggested navigation policies. In Figure I.1, we show a graphical representation of a navigation policy generated by AutoNav for `ebay.com`.

I.2.2 Server-side

The challenge with server-side security, is to ensure that there are no security vulnerabilities in the web application, as opposed to the browser, which was the focus of client-side security.

From the developer's perspective, this means writing code without bugs that attackers can exploit. However, writing bug-free code is hard, as is evident by the billions of credentials that have been stolen over the years due to poor security [9].

Many possible vulnerabilities can be present in web applications. OWASP's top 10 list [14] is a collection of the most critical web application vulnerabilities, with vulnerabilities ranging from injection attacks to authentication misconfiguration and XSS.

XSS vulnerabilities are caused by a web application reflecting user input as HTML code. Consider a forum where users, more formally an injection attacker (Section I.1), can post messages. If a user post `<b>hello</b>` and this is directly added to the HTML code produced by the application then the browser will interpret the `<b>` tag as HTML. This becomes more nefarious if the user posts a message containing `<script>` tags, as this allows them to execute JavaScript as the application. Using JavaScript attackers can steal cookies and other valuable information.

What makes XSS hard to detect in practice is that it is hard to know when the data should be escaped for HTML. If the data is read from a database it can be hard to determine for a developer if a malicious user could control that data. This makes finding the vulnerabilities the major challenge in mitigating XSS attacks.

To automatically find these vulnerabilities, web scanners can be used. These scanner try to interact with a web application with the goal of detecting a vulnerability. However, crawling a modern web application presents many challenges. Modern web applications have complicated workflows where combinations of links, form submissions, and JavaScript actions are required, as shown in the Figure I.2.

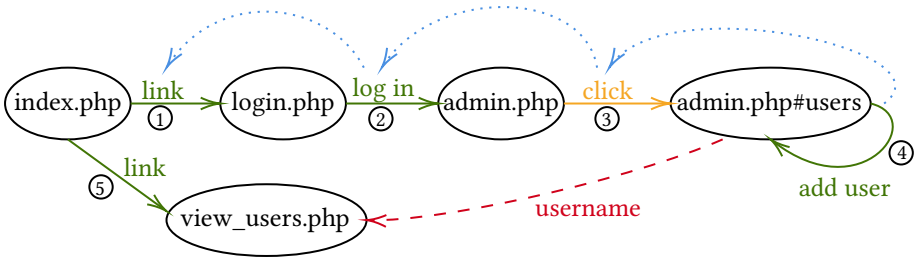


Figure I.2: Example of a web application where anyone can see the list of users and the admin can add new users. The dashed red line represents the inter-state dependency. Green lines are HTML5 and orange symbolises JavaScript. The dotted blue lines between edges would be added by our scanner to track its path. The sequence numbers shown the necessary order to find the inter-state dependency.

Additionally, the state of the web application is also important. For example, a prerequisite for adding a product review could be to add a product. In this case, the scanner would need to be able to add products before being able to test the security of the review functionality.

An orthogonal challenge all scanners face is solving input validation. It is common practice for web applications to validate the user's input. Common validations include ensuring phone numbers contain the correct number of digits, emails following the email format, correct formats for ZIP code, etc. A very useful and common attribute is "pattern". This attribute allows developers to specify regular expressions that are used as validation. For example, the pattern `.+@.+\.com` will match "any text" followed by an at-sign followed by "any text" and then finally ".com". This can be used to validate email addresses ending in ".com". However, generating input data that match these patterns is not trivial. And if the scanner is not able to provide valid inputs, they might not be able to fully explore the application and thus potentially miss vulnerabilities.

Background. Protecting against XSS is, in theory, simple. By converting the HTML tag characters < and > with the escaped values *<* and *>*, a large portion of XSS is solved. Depending on the precise context, other characters might need escaping too, for example, quotes (") can be escaped as `&quot;`. In the code below we see two examples where user input, `$name`, and `$url`, are reflected without any escaping. In these cases, an attacker could exploit this to gain JavaScript execution.

```
SERVER-SIDE CODE          =>  GENERATED HTML

Hello $name               =>  Hello <script>alert(1)</script>
<a href="$url">link</a>   =>  <a href="" onclick="alert(1)">link</a>
```

As it is easy to miss escaping data, it is also important to have methods to help find these vulnerabilities. One such method is to use automatic scanners. Automatic scanning can be divided into two categories: white-box and black-box. White-box analysis can be used if application artifacts, such as source code, models, and code

annotations, are available. In this case, the scanner can analyze these artifacts to uncover vulnerabilities. When these artifacts are not available, which is the standard case for penetration testing, black-box scanning can be utilized instead.

Black-box scanning dynamically interacts with the application, similar to how a user would. The scanner probes the application in different ways while analyzing the responses from the application for vulnerable patterns. For example, a black-box scanner can post `<script>alert(1)</script>` to a forum and analyze the response for a JavaScript alert message. If this is detected it would support the hypothesis that the forum has an XSS vulnerability. Previous research in the field of black-box scanners focused on one problem at a time. For example, the Jaek scanner focused on exploring JavaScript events [10]. The Enemy-of-the-state instead put the focus on modelling the state of the web applications [3]. And while these solutions made great strides in web scanning, scanners are still facing large challenges when scanning modern web applications.

Contribution. To improve the detection of vulnerabilities in web applications we explore new methods for interacting and crawling modern web application.

In Paper B [5], we present Black Widow where we combine the strengths of previous scanners while minimizing their weaknesses, in combination with novel detection methods for XSS vulnerabilities. This allows us to follow the complex paths in Figure I.2. We show that this increases the overall coverage of scanned web applications compared with previous scanners. And it allows us to find vulnerabilities in modern web applications.

As Black Widow, and all previous scanners, are unable to solve client-side validation patterns, we present Black Ostrich in Paper C [8]. Here we design a novel method for improving web scanners by leveraging string-based constraint solving, based on satisfiability modulo theories (SMT). Using this we shown that we can generate more valid input data than previous scanners, as well as data containing XSS payloads. For example, specific patterns like `.*France`, which requires the input to end with "France". This is something our scanner solves but all other fail on. In Figure I.3 we show the general approach to extend any web scanner with a SMT solver.

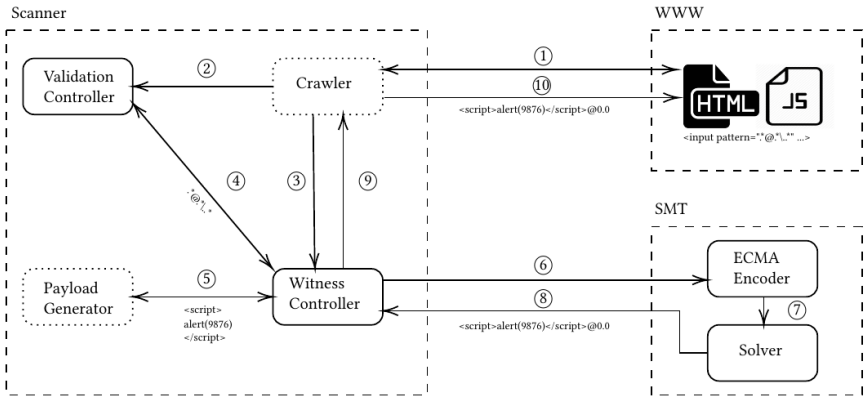


Figure I.3: Our general web scanner system architecture including both extended scanner and SMT solver. In Black Ostrich we combine the Black Widow scanner and Ostrich solver.

I.3 Browser extensions

The challenge with browser extensions is both detecting and ensuring that they are neither malicious nor vulnerable. Browser extensions live somewhere between the client-side (web browser) and server-side (web application). They are small programs that run inside the web browser with the power to interact with websites and network traffic on the user's behalf. For example, extensions might modify the aesthetics and accessibility of websites by changing colors and fonts. Extensions can also be password managers that can remember and generate passwords for users. There are also many popular extensions for blocking online advertisements and improving privacy by limiting network requests to some services. However, with these powerful capabilities, extensions can, and historically has, acted maliciously by stealing user information and modifying websites to inject more advertisements. Not only are malicious extensions a problem but so are vulnerable extensions. As these programs have powerful APIs, and store privacy-sensitive user data, ensuring that they are safe from attacks from web sites and extensions is important.

Analyzing extensions is not easy as there are many challenges for both static and dynamic approaches. Extension code can be minified or even obfuscated, making static analysis challenging. In addition, extensions can rely on code from remote servers, making static analysis impossible. While dynamic analysis can overcome this to some extent, here we are instead faced with "time bombs". During our research, we found multiple extensions that would wait between 30 minutes and five days before starting their malicious behavior.

Background. The main distributor of extensions is the Google Web Store. While Google is security vetting extensions before allowing them in the store there are still some that are able to slip through.

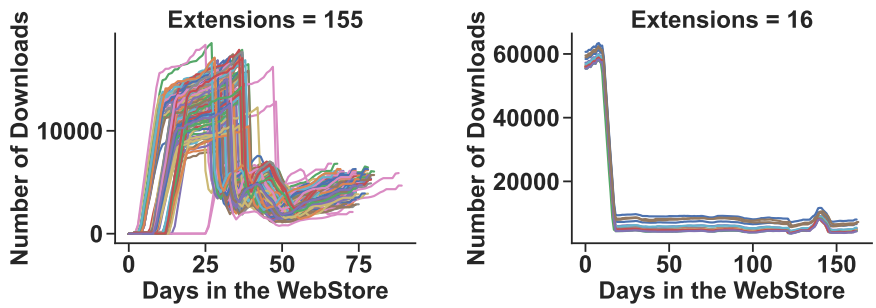
Table I.1: Summary of the attacks versus the ecosystem presented in this Paper D.

Attack	Subattack	Attacker	Victim	In wild
Password	Chrome autofill	Extension	Extension/User/Web page	Novel
	Virtual keyboard	Extension	Extension/User/Web page	Novel
Traffic		Extension	User	4,410
Inter-extension	Collusion	Extension	User	Benign
	History poisoning	Extension/Web page	Extension	1,349
	Code execution	Extension/Web page	Extension	1,349
	Fingerprint	Extension	User	10,785

Previous research in the field have also tried to improve the detection of both malicious and vulnerable extensions. To detect privacy leaks and other types of information stealing extensions, Mystique [2] uses taint analysis by modifying the V8 engine of Chromium. Moreover, there are also methods for detecting vulnerable extensions. Somé [13] used a combination of manual and static analysis to detect code execution vulnerabilities in extensions. While these methods are powerful they still focus on a narrow part of the problem that needs to be expanded to cover more types of attacks, vulnerabilities and detection methods.

Contribution. While previous methods focused mainly on code analysis to detect specific cases of malicious *or* vulnerable extensions, we want to extend the threat model to contain both malicious *and* vulnerable extensions. Therefore, in Paper D [6] we harden the security analysis of web extensions by extending the threat model. We also systematically analyze the extension ecosystem and map the possible interactions between the three agents: users, websites, and extensions. Based on the systematical analysis we combine both static and dynamic analysis to analyze a set of over 130,000 extensions. As a result, we find thousands of both vulnerable and malicious extensions. Our systematic analysis also led us to develop a novel active password stealing attack. We summarize these findings in Table I.1.

To combat the shortcomings of static and dynamic code analysis, we propose a novel method in Paper E [11] that can find malicious extensions without any source code analysis. Instead, our method analyzes the changes in downloads on the Chrome Store over multiple months. We show that these "download patterns" can be used to detect similarities between extensions, including maliciousness. Figure I.4(a) illustrates a cluster of 155 extensions that follow a similar pattern. In this case 154 of those extensions had similar code and were malicious. Conversely, we can also cluster similar but benign extensions, as shown in Figure I.4(b). While it is hard to determine exactly why clusters of extensions follow similar patterns, we believe it can be related to malicious developers paying for fake downloads to make extensions seem more popular. We have also seen extensions being used as part of other malware campaigns.



(a) Malicious TabHD extensions

(b) Benign FreeAddon extensions

Figure I.4: Download patterns for one malicious clusters and one benign.

I.4 Embedded Systems

The challenge with embedded systems is that as they become more complex and allow for third-party code to run, they too need to ensure client-side security. A great example of such a system is Android. Android is a popular operating system for mobile phones. While Android is mainly developed by Google, it also allows third-party developers to create and distribute apps on the Google Play store. By allowing third-party apps users can quickly create and share their favorite apps without having to wait for the first-party company to develop them.

The downside with third-party apps is that it is hard to ensure that the apps are not malicious or simply poorly implemented and vulnerable.

Background. Google is currently developing a new version of Android, named Android Automotive, which will run in the infotainment systems of cars. Android Automotive provides an excellent chance to research the security implications of porting a relatively secure platform, Android on phone, to work in a new ecosystem of embedded systems in cars. Similar to Android, Android Automotive uses a permission model. This forces apps to define which permission they want to use before being installed. For example, permission to use the camera or read the user's location. The interesting thing with Android Automotive is all the potential new vehicle-specific APIs and permissions. While these might be enough for normal Android, the move to the vehicle domains requires a new thorough analysis of the security assumptions.

Contribution. In Paper F [4] we analyze new automotive permission model and its APIs. We found three categories of attacks, disturbance, availability, and privacy. Disturbance attacks are a novel vector since it targets a new asset, the attention of the driver. Android was not designed with this in mind since it is not critical what the user focuses on. To counter this we develop AutoTame, a set of static analysis methods that can detect Android Automotive apps using dangerous APIs. We also present availability attacks that can crash the navigation in the infotainment

system. Here we also suggest OS-level mitigations. finally, In our research, we found permission-less exfiltration methods using the default music player. We also found new methods for acquiring sensitive information. In our case study of the Android Automotive Spotify app our improved information flow analysis can detect implicit leaks based on location, as shown in Figure F.3.

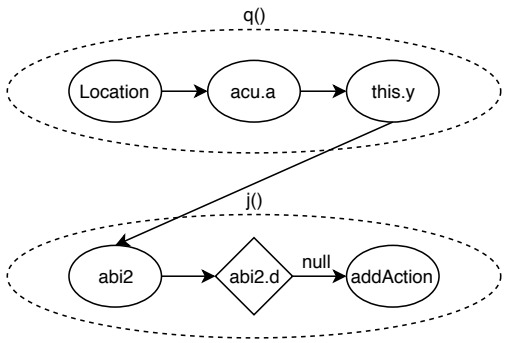


Figure I.5: The publicly observable addAction function is implicitly dependent on the private location information.

Thesis structure

This thesis comprises a collection of six papers. The first three (A,B,C) are related to client-side and server-side web security. The next two (D,E) focuses on browser extensions and the final one (F) targets embedded systems.

Part 1

Paper A **AutoNav: Evaluation and Automatization of Web Navigation Policies** This paper performs a first investigation of the new `navigate-to` CSP directive. We systematically analyze the potential vulnerabilities introduced by `navigate-to` with respect to the full web ecosystem.

Paper B **Black Widow: Blackbox Data-driven Web Scanning** In this paper we explore methods for improving the state-of-the-art in web application vulnerability scanning. We analyze the main challenges black-box scanners face in terms of vulnerability detection and code coverage.

Paper C **Black Ostrich: Web Application Scanning with String Solvers** In this paper we continue to effort to improve web scanners from Paper B but here focus on the orthogonal problem of solving input validation. We present challenges and solutions to both extracting input validation methods and solving highly complex regular expressions in the form of input validation patterns.

Part 2

Paper D **Hardening the Security Analysis of Browser Extensions** In this paper we explore the web browser extension security ecosystem and threat model. We research new static and dynamic methods for analysis and detection of both malicious and vulnerable extensions.

Paper E **No Signal Left to Chance: Driving Browser Extension Analysis by Download Patterns** From the challenges faced in the code analysis in Paper D, we explore new meta-data analysis methods in this paper. We explore the use of machine learning algorithms to cluster extensions with

similar meta-data and test if these clusters can be correlation to malicious behavior in the extension.

Part 3

Paper F **On the Road with Third-Party Apps: Security Analysis of an In-Vehicle App Platform** This paper aims to uncover new security vulnerabilities and attack vectors from the porting of Android to the vehicle-specific Android Automotive. We systematically investigate the attack surface of locally running third-party apps in vehicles.

III

Statement of contributions

This chapter lists the abstracts of the individual chapters and outlines the personal contributions for each.

A AutoNav: Evaluation and Automatization of Web Navigation Policies

Benjamin Eriksson, Andrei Sabelfeld

Undesired navigation in browsers powers a significant class of attacks on web applications. In a move to mitigate risks associated with undesired navigation, the security community has proposed a standard that gives control to web pages to restrict navigation. The standard draft introduces a new `navigate-to` directive of the Content Security Policy (CSP). The directive is currently being implemented by mainstream browsers. This paper is a first evaluation of `navigate-to`, focusing on security, performance, and automatization of navigation policies. We present new vulnerabilities introduced by the directive into the web ecosystem, opening up for attacks such as probing to detect if users are logged in to other websites or have active shopping carts, bypassing third-party cookie blocking, exfiltrating secrets, as well as leaking browsing history. Unfortunately, the directive triggers vulnerabilities even in websites that do not use the directive in their policies. We identify both specification- and implementation-level vulnerabilities and propose countermeasures to mitigate both. To aid developers in configuring navigation policies, we develop and implement AutoNav, an automated black-box mechanism to infer navigation policies. AutoNav leverages the benefits of origin-wide policies in order to improve security without degrading performance. We evaluate the viability of `navigate-to` and AutoNav by an empirical study on Alexa's top 10,000 websites.

Statement of contributions This was a collaboration with Andrei Sabelfeld. Benjamin was responsible for analyzing the potential vulnerabilities, designing and implementing AutoNav, and performing the evaluation.

Appeared in: *Proceedings of the Web Conference (WWW), 2020.*

B Black Widow: Blackbox Data-driven Web Scanning

Benjamin Eriksson, Giancarlo Pellegrino, Andrei Sabelfeld

Modern web applications are an integral part of our digital lives. As we put more trust in web applications, the need for security increases. At the same time, detecting vulnerabilities in web applications has become increasingly hard, due to the complexity, dynamism, and reliance on third-party components. Blackbox vulnerability scanning is especially challenging because (i) for deep penetration of web applications scanners need to exercise such browsing behavior as user interaction and asynchrony, and (ii) for detection of nontrivial injection attacks, such as stored cross-site scripting (XSS), scanners need to discover inter-page data dependencies.

This paper illuminates key challenges for crawling and scanning the modern web. Based on these challenges we identify three core pillars for deep crawling and scanning: navigation modeling, traversing, and tracking inter-state dependencies. While prior efforts are largely limited to the separate pillars, we suggest an approach that leverages all three. We develop Black Widow, a blackbox data-driven approach to web crawling and scanning. We demonstrate the effectiveness of the crawling by code coverage improvements ranging from 63% to 280% compared to other crawlers across all applications. Further, we demonstrate the effectiveness of the web vulnerability scanning by featuring no false positives and finding more cross-site scripting vulnerabilities than previous methods. In older applications, used in previous research, we find vulnerabilities that the other methods miss. We also find new vulnerabilities in production software, including HotCRP, osCommerce, PrestaShop and WordPress.

Statement of contributions This paper was written in collaboration with Giancarlo Pellegrino and Andrei Sabelfeld. Benjamin was responsible for developing the new scanning method, designing and implementing the method in Black Widow and performing the evaluation.

Appeared in: *Proceeding of the IEEE Symposium on Security & Privacy (IEEE S&P), 2021.*

C Black Ostrich: Web Application Scanning with String Solvers

Benjamin Eriksson, Amanda Stjerna, Riccardo De Masellis, Philipp Ruemmer, Andrei Sabelfeld

Securing web applications remains a pressing challenge. Unfortunately, the state of the art in web crawling and security scanning still falls short of deep crawling. A major roadblock is the crawlers' limited ability to pass input validation checks when web applications require data of a certain format, such as email, phone number, or zip code. This paper develops Black Ostrich, a principled approach to deep web

crawling and scanning. The key idea is to augment web crawling, based on the Black Widow tool, with string constraint solving to dynamically infer suitable inputs from regular expression patterns in web applications, and thereby pass input validation checks. To enable this use of constraint solvers, we develop new automata-based techniques to handle complex real-world regular expressions, including support for the relevant features of ECMA JavaScript regular expressions, and implement those methods in the Ostrich solver. We evaluate Black Ostrich on a set of 8 821 unique validation patterns gathered from over 21 667 978 forms from a combination of the July 2021 Common Crawl and Tranco top 100K. For these forms and reconstructions of input elements corresponding to the `patterns`, we demonstrate that Black Ostrich achieves a 99% coverage of the form validations compared to an average of 36% for the state-of-the-art scanners, while also yielding a 45% increase for vulnerability detection. We further show that our approach can boost coverage by evaluating it on three open-source applications. Our empirical studies include a study of email validation patterns, simultaneously demonstrating that our regular expression encoding is practical, where we find that many (213/825) of the patterns are susceptible to trivial XSS injection attacks.

Statement of contributions Benjamin was responsbile for developing the general approach to extend scanners to communicate with SMT solvers and in particular extending Black Widow to communicate with Black Ostrich. In addition, Benjamin created the testbed for evaluating scanners abaility to solve client-side verification, including crawling Tranco for patterns. Finally, Benjamin analyzed open-source projects from GitHub and evaluated the scanners on them.

Appeared in: *Manuscript*

D Hardening the Security Analysis of Browser Extensions

Benjamin Eriksson, Pablo Picazo-Sanchez, Andrei Sabelfeld

Browser extensions boost the browsing experience by a range of features from automatic translation and grammar correction to password management, ad blocking, and remote desktops. Yet the power of extensions poses significant privacy and security challenges because extensions can be malicious and/or vulnerable. We observe that there are gaps in the previous work on analyzing the security of browser extensions and present a systematic study of attack entry points in the browser extension ecosystem. Our study reveals novel password stealing, traffic stealing, and inter-extension attacks. Based on a combination of static and dynamic analysis we show how to discover extension attacks, both known and novel ones, and study their prevalence in the wild. We show that 1 349 extensions are vulnerable to inter-extension attacks leading to XSS. Our empirical study uncovers a remarkable cluster of "New Tab" extensions where 4 410 extensions perform traffic stealing attacks. We suggest several avenues for the countermeasures against the uncovered

attacks, ranging from refining the permission model to mitigating the attacks by declarations in manifest files.

Statement of contributions Benjamin was responsbile for analyzing the extension ecosystem to unify the threat model for both malicious and vulnerable extensions. Benjamin also implemented static and dynamic methods for detecting Collusion, history poisoning and code execution attacks. Finally, Benjamin designed the new password stealer attack.

Appeared in: *ACM Symposium On Applied Computing (SAC), April 2022.*

E No Signal Left to Chance: Driving Browser Extension Analysis by Download Patterns

Pablo Picazo-Sanchez, Benjamin Eriksson, Andrei Sabelfeld

Browser extensions are popular small applications that allow users to enrich their browsing experience. Yet browser extensions pose security concerns because they can leak user data and maliciously act on behalf of the user. Because malicious behavior can manifest dynamically, detecting malicious extensions remains a challenge for the research community, browser vendors, and web application developers. This paper identifies download patterns as a useful signal for analyzing browser extensions. We leverage machine learning for clustering extensions based on their download patterns, confirming at a large scale that many extensions follow strikingly similar download patterns. Our key insight is that the download pattern signal can be used for identifying malicious extensions. To this end, we present a novel technique to detect malicious extensions based on the public number of downloads in the Chrome Web Store. This technique fruitfully combines machine learning with security analysis, showing that the download patterns signal can be used to both directly spot malicious extensions and as input to subsequent analysis of suspicious extensions. We demonstrate the benefits of our approach on a dataset from a daily crawl of the Web Store over 6 months to track the number of downloads. We find 135 clusters and identify 61 of them to have at least 80% malicious extensions. We train our classifier and run it on a test set of 1,212 currently active extensions in the Web Store successfully detects 326 extensions as malicious solely based on downloads. Driven by the download pattern signal, our code similarity analysis further reveals 6,579 malicious extensions.

Statement of contributions Benjamin design the dynamic scanning method for detecing query stealing extensions, which was used as ground truth for the machine learning. Benjamin also contributed to analyzing and verifying the results. Finally, Benjamin was responsible for the comparison with other methods for detecing malicious extensions.

Appeared in: *Manuscript*

F On the Road with Third-Party Apps: Security Analysis of an In-Vehicle App Platform

Benjamin Eriksson, Jonas Groth, Andrei Sabelfeld

Digitalization has revolutionized the automotive industry. Modern cars are equipped with powerful Internet-connected infotainment systems, comparable to tablets and smartphones. Recently, several car manufacturers have announced the upcoming possibility to install third-party apps onto these infotainment systems. The prospect of running third-party code on a device that is integrated into a safety critical in-vehicle system raises serious concerns for safety, security, and user privacy. This paper investigates these concerns of in-vehicle apps. We focus on apps for the Android Automotive operating system which several car manufacturers have opted to use. While the architecture inherits much from regular Android, we scrutinize the adequateness of its security mechanisms with respect to the in-vehicle setting, particularly affecting road safety and user privacy. We investigate the attack surface and vulnerabilities for third-party in-vehicle apps. We analyze and suggest enhancements to such traditional Android mechanisms as app permissions and API control. Further, we investigate operating system support and how static and dynamic analysis can aid automatic vetting of in-vehicle apps. We develop AutoTame, a tool for vehicle-specific code analysis. We report on a case study of the countermeasures with a Spotify app using emulators and physical test beds from Volvo Cars.

Statement of contributions This paper was in collaboration with Jonas Groth and Andrei Sabelfeld. Benjamin was responsible for finding and evaluating the attacks, designing the countermeasures and creating AutoTame.

Appeared in: *Proceedings of the International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS), 2019.*

Bibliography

[1] ars Technica. Equifax website borked again, this time to redirect to fake flash update, 2017. https://arstechnica.com/information-technology/2017/10/equifax-website-hacked-again-this-time-to-redirect-to-fake-flash-update/.

[2] Q. Chen and A. Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *CCS*, page 1687–1700, 2018.

[3] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *USENIX Security Symposium 12*, pages 523–538, 2012.

[4] B. Eriksson, J. Groth, and A. Sabelfeld. On the Road with Third-Party Apps: Security Analysis of an In-Vehicle App Platform. In *International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS)*, 2019.

[5] B. Eriksson, G. Pellegrino, and A. Sabelfeld. Black Widow: Blackbox Data-driven Web Scanning. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[6] B. Eriksson, P. Picazo-Sanchez, and A. Sabelfeld. Hardening the Security Analysis of Browser Extensions. In *SAC*, 2022.

[7] B. Eriksson and A. Sabelfeld. AutoNav: Evaluation and Automatization of Web Navigation Policies. In *Web Conference (WWW)*, 2020.

[8] B. Eriksson, A. Stjerna, R. D. Masellis, P. Ruemmer, and A. Sabelfeld. Black Ostrich: Web Application Scanning with String Solvers. In *Manuscript*, 2022.

[9] Information is beautiful. World's biggest data breaches & hacks, 2020.

[10] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *International Symposium on Recent Advances in Intrusion Detection*, pages 295–316. Springer, 2015.

[11] P. Picazo-Sanchez, B. Eriksson, and A. Sabelfeld. No Signal Left to Chance: Driving Browser Extension Analysis by Download Patterns. In *Manuscript*, 2022.

[12] P. D. Ryck, L. Desmet, F. Piessens, and M. Johns. *Primer on Client-Side Web Security*. Springer, 2014.

[13] D. F. Somé. Empoweb: Empowering web applications with browser extensions. In *S&P*, pages 227–245, 2019.

[14] The OWASP Foundation. Owasp top 10 - 2017, 2017.

[15] M. West. Content security policy level 3, 2018.

[16] M. West, A. Barth, and D. Veditz. Content security policy level 2, 2016.

Client-side and Server-side Web Security

A

AutoNav: Evaluation and Automatization of Web Navigation Policies

Abstract. Undesired navigation in browsers powers a significant class of attacks on web applications. In a move to mitigate risks associated with undesired navigation, the security community has proposed a standard that gives control to web pages to restrict navigation. The standard draft introduces a new `navigate-to` directive of the Content Security Policy (CSP). The directive is currently being implemented by mainstream browsers. This paper is a first evaluation of `navigate-to`, focusing on security, performance, and automatization of navigation policies. We present new vulnerabilities introduced by the directive into the web ecosystem, opening up for attacks such as probing to detect if users are logged in to other websites or have active shopping carts, bypassing third-party cookie blocking, exfiltrating secrets, as well as leaking browsing history. Unfortunately, the directive triggers vulnerabilities even in websites that do not use the directive in their policies. We identify both specification- and implementation-level vulnerabilities and propose countermeasures to mitigate both. To aid developers in configuring navigation policies, we develop and implement AutoNav, an automated black-box mechanism to infer navigation policies. AutoNav leverages the benefits of origin-wide policies in order to improve security without degrading performance. We evaluate the viability of `navigate-to` and AutoNav by an empirical study on Alexa's top 10,000 websites.

A.1 Introduction

As the power of the web platform grows, attackers increasingly target *client-side vulnerabilities* [3, 9, 12, 16, 18, 37, 39, 43, 50, 56, 57]. Exploiting these vulnerabilities is effective because clients manipulate highly sensitive information, like login credentials, banking, health, and location data, on behalf of the user.

A.1.1 Motivation

One of the bigger classes of client-side security vulnerabilities on today's web is *cross-site scripting (XSS)* [45]. An XSS vulnerability gives an attacker the power to

execute JavaScript code on another website. This can be used to steal user credentials, change the behavior of the application or render the website unusable. A common approach to mitigate this problem is to let servers send extra security policies along with each HTTP response. The web browser will then enforce these policies, for example, by restricting which scripts to allow on the webpage. These security policies have been defined by the web security community as part of *Content Security Policy (CSP)* [63].

Navigation attacks The current CSP standard (level 2 [63]) does not address attacks via *navigation*. Attackers can thus freely redirect users to malicious or inappropriate websites. This type of attack can affect the confidentiality, integrity and availability of the attacked website. For confidentiality, an attacker with injection capabilities can inject the following script to leak the secret cookie.

```
<script>
window.location = "http://evil.com/?c="+document.cookie;
</script>

```

When the script is executed the user will be sent to `http://evil.com`, along with their cookies, potentially allowing the attacker to take over their account. In addition to only stealing the cookie, the attacker could launch a phishing attack by designing `http://evil.com` to look like the attacked website. Here the user could be asked to supply more confidential information or be forced to download malicious software. The availability of the website is also compromised as every user visiting the page containing the injected script will be sent away. Note that while CSP can block scripts, an attacker could also force the user to perform a navigation by using meta tags as shown below. While not valid HTML, modern browsers will follow meta redirects in the HTML body.

```
<meta http-equiv="refresh"
      content="0;URL='http://evil.com/'" />

```

The `navigate-to` directive To mitigate these problems the World Wide Web Consortium (W3C) has drafted a standard for the new CSP directive `navigate-to` [61]. This directive has already been implemented in Chrome [36] and Firefox [28]. A common motivation for the directive is to increase the security on websites, as well as, give advertising platforms better control over navigations in ads [40]. We illustrate this in two example scenarios: HTML/JavaScript injection and malicious advertisement.

HTML/JavaScript injection Understanding the space of navigation links on a website can improve security thanks to `navigate-to`. By limiting the possible navigations, attackers will not be able to redirect users. A real-world example of where this policy would have helped is a vulnerability on `blockchain.info` [27]. Attackers were able to inject HTML and JavaScript into the search function on the page. This meant that a URL similar to `blockchain.info/?search=<code>`, which appears to point to `blockchain.info`, could redirect the user to another website. This is known as a reflective XSS vulnerability [48], as the code in the URL is reflected

onto the page. Although `blockchain.info` used CSP to mitigate XSS, it was still possible to inject HTML code that forces a redirect. With the new directive, the following CSP policy can mitigate this type of attack. This policy blocks any navigation attempt to anything but `self`, i.e. `blockchain.info`.

```
navigate-to 'self'

```

Malicious advertisement Advertisement platform providers benefit from ensuring that users who click on their ads end up on the correct page. This is especially important if the pages where the ads are served are sensitive to inappropriate material, e.g. websites for kids, governments, or highly respected financial websites. Using the new directive, advertisers would be able to block navigations leading to incorrect ads. The policy is required because even if the target site for the ad is correct when the ad is bought, the website can at a later stage be hacked or misconfigured. Google Ads could, for example, serve the following policy with an ad from `shoes.com`. This would only allow navigation to `https://shoes.com`, blocking both the HTTP version, as well as, possible deeplinks to apps like `app://shoes.com`. The `unsafe-allow-redirects` keyword allows for any number of server-side redirections before reaching `shoes.com`.

```
navigate-to  https://shoes.com 'unsafe-allow-redirects'

```

A.1.2 Research questions

The standardization [35, 61] and implementation [28, 36] efforts for `navigate-to` are well underway. The time is critical to ask questions on the security, performance, and adoptability of the proposed directive, before its adoption starts on the web. (Our analysis at the time of the writing confirms that the landing pages of Alexa's top 10,000 domains are yet to contain `navigate-to` CSP headers). By pursuing these questions, our goal is to deepen understanding of navigation policies and their impact, contribute to the emergence of the new standard, and to utilize our findings for settling the ongoing discussions by the community [29].

Security While there seems to be much to gain from a navigation policy, what is the impact on the security of the entire web ecosystem? For a fully-fledged security evaluation, we seek to uncover both new vulnerabilities and amplifying effects of known vulnerabilities. Our methodology is thus to investigate possibilities of exploiting the directive by a comprehensive range of attackers defined in the security literature [39]: injection [5], gadget [6], web [2] and passive network [25] attackers. Even though these attackers share some capabilities, they each have unique abilities, e.g. reading network traffic or hosting websites, and as such require individual analysis. This brings us to the questions of security: *Does the new policy "break the web"? Does the new policy introduce security vulnerabilities? How can they be mitigated and by whom?*

Automatization Once the new directive is secured, how can we aid its adoption? CSP has been notoriously hard to adopt, introducing insecure policies or broken websites [56, 57]. To help developers use the new directive, and increase both usability and adoptability, we investigate the possibility of automatically generating navigation policies. Hence, the question: *Can automatic mechanisms be used to help generate the new policy?*

Performance In contrast to CSP directives like `script-src`, intended to whitelist scripts that can be loaded by a webpage, the `navigate-to` directive will whitelist possible navigations. This results in already lengthy response headers becoming even larger, further increasing the overhead of security headers. This brings us to the question of performance: *What are efficient methods for delivering the new policy?*

A.1.3 Contributions

This paper is a first systematic evaluation of `navigate-to`. Our goal is to both initiate research on navigation security and to affect the emerging standards for navigation policies. We examine the security implications, efficiency, and the possibility of automatic generation of the new `navigate-to` policy.

Security The intricate connections between policies together with the growing complexity of the web results in new mechanisms becoming more challenging to incorporate into the ecosystem. This motivates the need to analyze multiple types of attackers, as well as, reexamining existing mechanisms in combination with new ones. We follow a methodology of examining the effects of `navigate-to` on a comprehensive range of attackers: injection [5], gadget [6], web [2] and passive network [25] attackers. By scrutinizing the full attack surface of the new directive, with respect to different types of attackers, we identify specification- and implementation-level vulnerabilities that can be exploited (Section A.3). The vulnerabilities allow attackers to probe other websites to detect if users are logged in or have active shopping carts, bypass blocking mechanisms of third-party cookies, leak browsing history, and open up new methods for exfiltration. This demonstrates that the directive "breaks the web" in the sense of introducing vulnerabilities even in otherwise secure websites that do not use the directive in their policies. We present mitigations to security problems, both for web and policy developers (Section A.4).

Automatization Looking ahead when the proposed mitigations are in place, our goal is to aid in the adoption of `navigate-to`. We develop AutoNav, an automatic mechanism for navigation policy inference (Section A.5). AutoNav crawls websites and generates `navigate-to` policies. The goal of this mechanism is to simplify the deployment of the new directive by helping web developers and security engineers to find fitting policies for their websites. To further improve security, AutoNav can also generate origin-wide policies for the new origin policy delivery mechanism that is currently being drafted [59]. This improves security by applying the policy to the entire origin, covering pages that are easy to forget, like error pages. We implement and evaluate the mechanism by an empirical study (Section A.6). In our experiments, we craw 100 pages per domain for 10,000 domains. Based on a subset

of 80 pages, AutoNav generates a policy for the remaining 20 pages. For 42% of websites, AutoNav generated a policy which fully covered the 20 pages, and at 59% 19 out the 20 pages were covered. Further investigation into the category of websites shows that shopping websites and adult websites are the easiest to cover.

Performance To evaluate the performance impact of the policy we perform an empirical study (Section A.6). Based on 10,000 crawled domains from Alexa's top 10,000, the policy will result in an overhead of 215 bytes for each HTTP response. We create simplification strategies to find a balance between security, performance and maintainability. These simplifications convert complicated policies with multiple subdomains to more manageable policies by using wildcards. For example, instead of including all language-specific subdomains from Wikipedia `navigate-to *.wikipedia.com` would be enough. Our simplification algorithm decreased the overhead by between 40% and 47%. Furthermore, we show that the use of an origin policy would result in an overhead of 1904 bytes in total, as opposed to per HTTP response. This is further decreased to 1004 bytes by using our simplification algorithm. A 900 byte reduction might not seem like much, but it can have a big impact on larger websites [21].

A.2 Background

Setting the background, we present the threat model in terms of relevant attackers. We describe CSP and how it relates to the origin policy. Finally, we explain navigation methods and how they are treated in the `navigate-to` directive.

A.2.1 Threat model

The main goal of the `navigate-to` directive is to give web developers control over where users can navigate from their website. The assets that need protecting include confidentiality, integrity and availability. Previous research has already shown how confidential information, such as cookies, can be exfiltrated using navigation [65]. While the new directive is a step in the right direction to address data exfiltration, Zalewski [65] points out that control over navigation is not necessarily enough. Attackers could, for example, inject HTML or JavaScript that change documents from private to public on a website like Dropbox. Forced navigation can also be used for phishing attacks by redirecting users to a similar-looking, but attacker-controlled, website.

Modern web browsers support many different methods for navigation, e.g. by clicking on a link, submitting a form, etc. These navigation methods, and the subset that the `navigate-to` directive is intended to apply to, are explained in Sections A.2.4 and A.2.5.

As mentioned above, we are interested in a comprehensive security evaluation of the impact of the directive on the entire web ecosystem. Hence, our threat model includes four types of attackers from the security literature [39]: injection, gadget, web and network attackers. In practice there is some overlap between the classes,

for example, an attacker with *web attacker* capabilities will usually also have *injection attacker* capabilities. However, the best mitigation strategy might be different depending on which specific class we need to defend against. Therefore it is important to study each distinct class of attacker.

Injection attacker The *injection attacker* [5] is able to inject content into a website. A typical example is a user who can post content on a forum. If the user's post contains JavaScript then that code could be executed by other users on the site, in this scenario, with the goal to force a navigation.

Gadget attacker The *gadget attacker* [6] is similar but more powerful as they are allowed to host code, or gadgets, on other websites. A notable example is JQuery which is a JavaScript snippet that is used by many websites. Since JavaScript do not support any isolation, these gadgets run with the same capabilities as other scripts on the website. A malicious gadget could exfiltrate information from the website it is integrated to, modify content on pages or even navigate the user away from the website.

Web attacker The *web attacker* [2] is able to host and configure a full website. This is especially important for advertisers who want to ensure that the landing page does not redirect to anything other than what was specified in the ad.

Passive network attacker A *passive network attacker* [25] can listen in on all the traffic sent from and to a client but can not decrypt HTTPS. If the traffic is not encrypted, the attacker can read passwords and session cookies being sent to the server.

Note that `navigate-to` is not designed to handle network attacks. Yet we pay attention to network attackers in our effort to analyze the impact of the directive on the entire web ecosystem.

A.2.2 CSP

CSP is intended to mitigate cross-site scripting (XSS) and other code injection attacks. The current version of CSP, level 2, is supported by all major web browsers [26]. Level 3, which includes the new `navigate-to` directive, is being discussed and drafted [61].

CSP protects the users by specifying which resources and scripts are allowed on a page. The web server sends the CSP policies each time a user requests a page. These policies are then enforced by the browser to, among other things, block XSS. The policy below will only allow scripts to be loaded from the current origin, still blocking any injected inline scripts. In addition, the reporting header `Content-Security-Policy-Report-Only` [33] can be used to report policy violations without enforcing them. These reports are sent as POST requests to the server. They can also be detected using `SecurityPolicyViolationEvent` in JavaScript.

```
1 Content-Security-Policy: script-src 'self'
2
```

A.2.3 Origin policy

Today, CSP headers are sent with every HTTP(S) response, which is a concern for both safety and performance [50]. For security, it is easy to forget the policy on special pages, like error pages [59]. It also harms performance because servers need to repeat the same policy for each response, even if the policy should apply to all. To address this, specifications are being drafted [59], implemented [60], and evaluated [50] to enable *origin-wide policies*, known as *origin policies* [59] or *origin manifests* [50]. Using an origin policy, the server only needs to include once which policies should apply to the whole origin.

A.2.4 Navigation

Navigations can be performed in many different ways by browsers, e.g. by clicking on a link, submitting a form or running JavaScript. Navigation methods can be split into two different categories, user-initiated or document-initiated. While navigation is defined in the Fetch [52] and HTML [4] standards, the exact methods available depend on the web browser implementation. We make an effort to summarize the most common methods in Table A.1. The *Automatic* column shows if the navigation method can be performed automatically. This is true for all JavaScript function and, in case JavaScript is allowed, `<a>` and `<form>` tags. It is worth noting that while a web page cannot read a user's browsing history, it can initiate navigation to go back or forward in the browser history. There are many `.location` functions in JavaScript that can navigate, e.g. `window`, `document`, `parent`, etc. They all use the Location object defined in the HTML standard [4]. Some functions, like `window.navigate`, only works in Internet Explorer [11]. The last column specifies which methods `navgiate-to` affects.

Table A.1: Navigation methods together with initiator and possibility to automatically navigate.

Method	Initiator	Automatic	Affected
`<a>` tag	Document	With JavaScript	✓
`<form>` tag	Document	With JavaScript	✓
`<meta>` tag	Document	Yes	✓
`<iframe>` tag [51]	Document	Yes	✓
`window.open` [53]	Document	Yes	✓
`*.location` [4]	Document	Yes	✓
`window.navigate` [11]	Document	Yes	✓
Typing the URL	User	No	
History buttons	User & Document	Yes	
Home button	User	No	

A.2.5 Navigate-to directive

The *navigate-to* directive gives developers the power to control the navigations a document can initiate. Document initiated navigations are discussed in Section A.2.4. This directive makes it harder for attackers to inject code to redirect users from legitimate websites. For example, if an attacker manages to inject links on `disney.com` then Disney's reputation is at stake if links lead to inappropriate websites. To tackle this, Disney could add the following to their CSP policy:

```
navigate-to *.disney.com *.thewaltdisneycompany.com
```

This would instruct the browser to only accept navigations to subdomains of `disney.com` and `thewaltdisneycompany.com`, and block all navigations to other websites. The standard also introduces the new keyword `unsafe-allow-redirects`, which allows any redirects as long as the final destination is allowed by the policy. It is deemed less safe since it does not have full control over all the sites in the redirect chain. However, it is still better than nothing in terms of limiting navigations.

The `navigate-to` directive is currently being standardized by W3C [61] and implemented in Chrome [36] and Firefox [28]. It is available in the current version of Chrome (version 77.0.3865), and other Chromium-based browsers like Edge and Brave, behind a flag that enables experimental features. It is also available in Firefox Nightly (Version 71.0a1) behind a flag [28].

A.3 Vulnerabilities

This section presents vulnerabilities and security concerns related to the `navigate-to` policy. These vulnerabilities are not navigation attacks, but rather vulnerabilities that become possible due to `navigate-to`. Except for the last vulnerability in Section A.3.3.3, where we rather want to show that a small improvement to `navigate-to` can solve an existing problem. The policy introduces new methods for acquiring privacy-sensitive information, circumvention of security mechanism and data exfiltration. All the attacks described in this section have been tested in practice. While some of the vulnerabilities, like the data exfiltration, relies on the existence of other vulnerabilities, like content injection, the `navigate-to` adds a new layer to the attacks. This possibility of combining attacks shows the importance of reexamining existing ones when introducing new mechanisms.

A.3.1 Methodology

To systematically find vulnerabilities we distinguish vulnerabilities relating to the specification of the `navigate-to` directive and vulnerabilities related to its implementation. For each category, we divide the investigation of vulnerabilities pertaining to confidentiality, integrity and availability, in accordance with the CIA triad. We draw on our threat model and examine vulnerabilities with respect to injection, web,

gadget, and passive network attackers. Finally, we analyze how the directive can be used to circumvent modern countermeasures, such as third-party cookie blocking.

The presentation of the vulnerabilities is ordered by our estimate of their impact, from high to low. Table A.2 lists the vulnerabilities we discover together with their corresponding attacker model. Interesting to note is that resource probing and Google Search profiling can be exploited to attack websites that themselves do not use the `navigate-to` directive. This results in previously security websites becoming insecure.

Table A.2: The uncovered vulnerabilities together with corresponding attacker model.

Vulnerability / Attacker	Injection	Web	Gadget	Passive network
Resource probing		✓		
Google Search profiling		✓		
Third-party cookie bypass		✓		
History sniffing		✓		
Data exfiltration	✓		✓	
Ads leaking data				✓

A.3.2 Specification

The following vulnerabilities are present in the specification. This means that any browser following the specification correctly will be vulnerable.

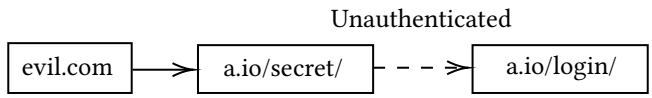


Figure A.1: A user visiting `evil.com` will be navigated to `a.io/secret/`. If they are not logged in, they are further redirected to `a.io/login/`.

A.3.2.1 Resource probing

In cases where web applications redirect based on sensitive resources, these resources could be probed. For example, probing for the existence of Dropbox files. The probing attacks in this section are deterministic, as opposed to other attacks that rely on timings [55]. The attacks are also general and could potentially be used on any website, not solely on advertiser platforms such as the attack presented by Venkatadri et al. [54].

A malicious website, i.e. a web attacker, can navigate a user to `dropbox.com/preview/wallet.txt` to detect if a user has a file named `wallet.txt`. If no such file exists then the user is redirected to `dropbox.com/home/wallet.txt`, making it possible to craft a policy which blocks `/preview/` but not the redirection to `/home/`, like the following. Note here that

we only use path-sensitivity to block `/preview/`. If we are redirected, then path-sensitivity is no longer available and we only have to allow `dropbox.com`. The main difference compared to previous work on CSP redirections [23] is that we only need path-sensitivity for the first request, not the redirects.

```
navigate-to 'unsafe-allow-redirects' https://www.dropbox.com/not_preview/;

```

By utilising invisible iframes multiple files can be checked in parallel, without the user being navigated away from the malicious website.

One specific application of resource probing that has been researched before is login detection. Previous methods [20, 32] relies on third-party cookies, which can be blocked by the user or by the proposed default SameSite policy [62]. Instead, note that a navigation to `facebook.com/settings/` will redirect the user to the login page, `facebook.com/login.php`, if they are not authenticated, similar to Figure A.1. By allowing only one of these URLs in the policy, the attacker can differentiate between a successful navigation and a blocked one. This feature makes our method more powerful and general.

We have also found that on some E-commerce websites it is possible to detect if a customer has anything in their shopping cart. This is because navigating directly to the shopping cart or checkout page sometimes redirects the user depending on the content of the cart. PrestaShop, which is an E-commerce platform used on hundreds of thousands of websites [8], does exactly this. By visiting `example.com/en/order` a user will be redirected to `example.com/en/cart`, assuming `example.com` uses PrestaShop.

Some of the probing attacks can leak more data if they are done in an active fashion. The PrestaShop attack can be improved to, in theory, enumerate the full cart. This is due to a Cross-Site Request Forgery (CSRF) [47] vulnerability in PrestaShop, currently being disclosed, which allows an attacker to add and remove items. Using this method an attacker can repeatedly remove items and then check if the cart is empty.

These are only a few examples we have found where redirects are based on sensitive data. We believe that many more such redirects currently exists on the web. Furthermore, navigations can bypass `lax` SameSite cookies, making the attack possible on sites where previous CSRF attacks were not possible.

A.3.2.2 Google Search profiling

Google Search relies on *personalized search* [19], meaning that the results of a search query are based on the users' previous interactions with Google. A recent study [24] shows that users are put into so-called "filter bubbles" by Google, resulting in varying result when searching for political terms such as "gun control" or "immigration". A web attacker can craft a malicious website which uses the `navigate-to` directive together with Google's *I'm feeling lucky* function to extract top results from visitors. This type of extraction attack is called cross-site search and has previously been successfully mounted against Gmail and other websites [17]. The main difference is that previous methods have relied on timing, whereas our method is fully deterministic. Castelluccia et al. [10] were also able to infer sensitive information

about users based on Google Searches. However, their approach required network attacker capabilities and assumed the traffic was unencrypted, which is not the case anymore. Our attack can be mounted by anyone with the capability to set up a website.

The attacker can then use these top results from Google to infer these filter bubbles. Using the URL `https://www.google.com/search?q=QUERY&btnI`, Google will automatically redirect the user the top result for term "QUERY". Therefore the *I'm feeling lucky* function acts as an open redirector, which is something both OWASP [46] and Google [34] themselves warn about. It is well known that Google has this problem but so far they choose to accept the risk [1]. However, `navigate-to` adds a new dimension to the problem as it enables attackers to infer data about users.

To exploit this the attacker can specify a report-only policy that only allows `google.com`, as shown below. The redirect will violate the policy and the browser will dutifully report which domain was in violation to the malicious website. The attacker can iteratively update the query to get more results. Assuming searching for "news" would return `news.com`, then the next query would be "news -site:news.com", which excludes `news.com` and perhaps returns `reports.com` instead. Another attack vector would be other search engines using this approach to directly copy personalized search results from Google, similar to what Bing did [41].

```
Content-Security-Policy-Report-Only: navigate-to 'unsafe-allow-redirects'
    google.com

```

A.3.2.3 Third-party cookie bypass

A cookie is a piece of data that websites can save locally on users' machines. [31] Depending on how the cookie is acquired, it will either be considered a first-party cookie or a third-party cookie. A navigation will result in first-party cookies while image request and similar results in third-party cookies.

Third-party cookies are useful for advertisers [14] as it allows them to use small tracking pixels [15] for tracking users. Modern browsers allow users to block third-party cookies or do it by default [42].

Previous work has demonstrated how Cookie Synchronization [7, 38] can be used by ad platforms to effectively break the same-origin policy. Privacy-aware users can mitigate this by blocking third-party cookies altogether. However, the `navigate-to` directive introduces a new method for advertisers to circumvent this by using navigations. As it requires control over the CSP headers, web attackers are the main threat. Figure A.2 shows a user visiting `a.io`, then being forcibly navigated to `track.com` and acquiring a first-party cookie. Using the following policy, the redirection will be blocked, making the attack unnoticeable to the user.

```
navigate-to 'unsafe-allow-redirects'

```

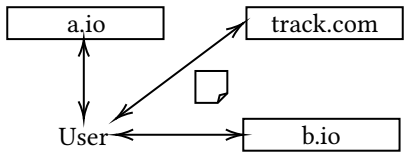


Figure A.2: When a user visits `a.io` or `b.io`, they can force the user to obtain first-party cookies from `track.com`.

A.3.3 Implementation

The following vulnerabilities are due to implementation decisions. We focus on Chrome's [36] and Firefox's [28] implementations of `navigate-to`,

A.3.3.1 History sniffing

The `navigate-to` policy can, in some cases, be exploited by a web attacker with a malicious website to probe which websites a user has visited. The attack uses the fact that websites using HSTS force the browser to remember and upgrade insecure connections. Previous methods exploiting this have relied on timing attacks which are now mitigated [64].

Using `navigate-to`, a malicious website can make a POST request to another site which uses HSTS but is not preloaded. If the site redirects based on the POST data then the attacker might be able to detect if a user has visited the site before. This is possible because if the user has visited the site before it will result in an internal redirect (HTTP 307), which keeps the POST data. Otherwise, the server will redirect (HTTP 301/302), which drops the POST data. If the server specifically performs a 307 redirect then the attack will not work. By crafting a CSP that does not allow the redirect, the attacker can differentiate between the two cases, denoted X and Y in Figure A.3. This is, for example, possible using the login function on the popular social media website VK.

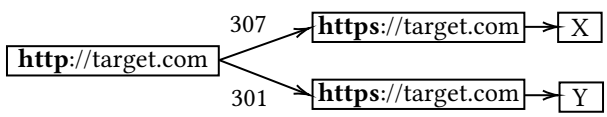


Figure A.3: If `target.com` uses HSTS, and the user has visited the site before, then the browser will automatically upgrade the connection to HTTPS using a 307 redirect instead of a server side 301.

A.3.3.2 Data exfiltration and communication

Previous research has shown that data exfiltration is possible in the face of CSP [49]. The usage of forms and links to exfiltrate data has also been studied [65]. However, the `navigate-to` policy introduces an improved method for exfiltration, and two-way communication, based on JavaScript together with navigation. This works in

Chrome, but not Firefox, as Chrome does not unload the page for `navigate-to` violations.

Consider a website using `connect-src 'none'` and `frame-src 'none'` to limit external loads as much as possible. The `connect-src` directive protects against some exfiltration methods including XHR, fetch and `<a ping>`, while `frame-src` will block exfiltration to iframes. Assume the website uses `unsafe-allow-redirects` followed by a list of allowed URLs. Note here that we show that *unsafe* has implications beyond the scope of restricting navigation. An attacker capable of injecting JavaScript, i.e. either an injection or gadget attacker, can now use `window.location`, as shown in the listing below, to exfiltrate arbitrary data. Each navigation request will exfiltrate data, then be blocked by the policy, as the attacker can choose a website outside the CSP whitelist. Furthermore, by adding a `SecurityPolicyViolation` event listener the attacker can inspect the blocked URI in the violation. To send a response, `evil.com` would redirect the request to a subdomain like `<msg>.evil.com`.

```
function exfiltrate (data) {
  window.location = "http://evil.com/?d=" + data;
}

```

The main difference between not using `navigate-to` and using the policy described is that by blocking the navigations, the control is returned to the attacker, allowing for further stealth exfiltration and communication.

A.3.3.3 Ads leaking data

We have found that ads served over HTTPS can still leak the final landing page to a passive network attacker if an ad in the redirection chain is unencrypted. While network-level eavesdropping is outside of CSP's threat model, the `navigate-to` directive presents a great opportunity to fix this problem. The problem stems from the fact that when a user clicks on an ad they can be channeled through multiple tracking websites. Listing A.1 shows a chain where the user is redirected to three different websites before the landing page. We performed a small empirical study using the same dataset as in Section A.6. We extracted all iframes and compared their source URL to a list of known advertisement platforms, e.g. DoubleClick. If the URL matched we followed it and recorded the redirects. This resulted in 24650 unique ads, of which 26.7% have a website between the advertisement platform and the landing page. This highlights the need for advertisement platforms to consider potential redirects from tracking websites and further motivates the need for the `navigate-to` directive.

```
https://www.googleadservices.com/...
http://www.kqzyfj.com/...
http://cj.dotomi.com/...
http://www.emjcd.com/...
https://<landing page>/...

```

Listing A.1: Example of an ad chain containing three different unencrypted domains between the encrypted ad platform and landing page.

As can be seen in Listing A.1, both the first and last websites use HTTPS but there exist sites between that are unencrypted. This is very hard for a user to detect as both the ad and the landing page seems secure. The problem with having HTTP in the chain is that an eavesdropper can follow the request and find the landing page. Our empirical experiments show 10.6% of the ads follow this pattern. As ads become more personal this becomes a privacy concern. Advertisements related to economic status or specific diseases might be leaked without the user's knowledge.

A.4 Countermeasures

This section presents countermeasures to the vulnerabilities in A.3. The countermeasures cover the specification, mitigations for web developers, as well as, implementation improvements in web browsers. Similarly to the vulnerabilities in Section A.3 we distinguish specification- and implementation-level countermeasures.

A.4.1 Specification

A.4.1.1 Resource probing

Previous login detection methods have forced web developers to rewrite their applications to avoid special types of redirections. As mentioned in [13], Google added an extra regex check to make sure the redirection did not lead to resources that could be loaded cross-origin, e.g. "jpg", "js" and "ico".

The `navigate-to` policy circumvents this by being able to block and report different paths in the URL, i.e. it is possible to block `example.com/settings/` and allow `example.com/login/`. If `/settings/` redirects to `/login/` for unauthenticated users, then the CSP report log can be inspected to discern between authenticated and unauthenticated users.

To fix this, path precision could be removed from the policy. If an origin as a whole can not be trusted, it seems to add little security to trust certain paths on the origin. Since these vulnerabilities affect websites that do not use `navigate-to`, we also present countermeasures web developers can implement. We recommend avoiding redirection based on secrets. Instead, by showing an error page or rendering the login form on the same page the website is guaranteed to not leak any data, as there will be no redirections. If redirection is necessary, encoding paths in GET parameters, e.g. from `example.com/files/` to `example.com/?path=/files/`, also mitigates the problem.

A.4.1.2 Google Search profiling

For vulnerabilities like Google Search profiling, as presented in Section A.3.2.2, the key countermeasure is to avoid open redirects [46]. One possible way for Google to accomplish this without removing the *I'm feeling lucky* function is to use a CSRF token [47].

A.4.1.3 Third-party cookie bypass

The navigation path through the redirection chain can depend on the user's cookies. For this reason, it is not possible to block cookies while checking if the navigation is allowed. Instead, we suggest that cookies attained during the check are temporarily sandboxed and then removed if the navigation is blocked.

A.4.2 Implementation

A.4.2.1 History sniffing

Privacy problems related to HTTP Strict Transport Security (HSTS) [22] has been researched before [44]. However, they focused on tracking mechanisms similar to cookies but harder to remove.

The solution is to ensure that an attacker can not differentiate between the paths in Figure A.3. Again, it becomes the web developers responsibility to either use an internal redirect or not redirect on post data.

A.4.2.2 Data exfiltration

What makes this attack extra powerful is its ability to regain execution control after the navigation fails. It is not specified what should happen when the `navigate-to` policy blocks a navigation attempt. Currently, Chrome seems to simulate a 204 response [58], resulting in the continuation of the script, and the possibility to exfiltrate more data. Firefox, on the other hand, uses a full-page error that unloads the original document. By using this strategy the script will stop executing, blocking further exfiltration. The attack can also be mitigated by avoiding `unsafe-allow-redirects`, as this will block the exfiltration during the pre-navigation check.

A.4.2.3 Ads leaking data

The `navigate-to` directive could block redirect chains which contain HTTP websites. Currently, the policy `navigate-to https:` allows navigation to any website using HTTPS. However, combined with `unsafe-allow-redirects` HTTP is allowed in the chain, as long as the landing page is HTTPS. One solution is to add a value `unsafe-allow-https-redirects` which would only allow redirection by HTTPS. A more general solution is to split the policy into `navigate-to` and `navigate-by`, where the latter would apply as long as the request is redirected. When no redirect is received, the landing page is checked against the `navigate-to` policy. By using this method, the following policy would allow any HTTPS redirections which lead to `https://example.com`.

```
navigate-to https://example.com
navigate-by https:

```

A.5 AutoNav

We present AutoNav, an automatic mechanism to aid web developers in inferring policies for their websites. The mechanism crawls the website and creates a map of where pages can navigate. This mapping is used to generate and simplify the policies. AutoNav can generate both per-page policies, where each page on a website gets its own policy, and origin-wide policies [59].

A.5.1 Inference

We use a key-value map from the crawler to infer the policies. The page is used as a key, and a list of all possible navigations from the page is used as a value. Listing A.2 shows an example.

```
{
  "example.com/a.html": [facebook.com, google.com],
  "example.com/b.html": [twitter.com, google.com]
}

```

Listing A.2: Example of a key-value map generate from crawling two pages on `example.com`

Using the key-value map, AutoNav can generate separate policies for each page on the website. This is shown in Listing A.3. AutoNav can also generate an origin-wide policy based on the union of all the URLs, as shown in Listing A.4. These policies are then simplified, using the method described in Section A.5.2, to reduce the size and improve maintainability.

```
{
  "a.html": "navigate-to facebook.com google.com",
  "b.html": "navigate-to twitter.com google.com"
}

```

Listing A.3: Per-page policies generated from Listing A.2.

```
{
  "*": "navigate-to facebook.com twitter.com google.com"
}

```

Listing A.4: Origin-wide policy generated from Listing A.2.

A.5.2 Policy generation

The navigation policy is a whitelist of URLs that the user is allowed to navigate to. In the most secure setting, the policy should contain the full URLs to each allowed target. While secure, this creates big and hard to maintain lists of URLs requiring much bandwidth. Take Wikipedia for example, their policy could consist of all subdomains like `en.wikipedia.org`, `es.wikipedia.org`, etc. for each language. A more compact policy is

`*.wikipedia.org`. This simplification results in both less data being transmitted and a more maintainable policy, however, it does decrease security as it also allows `evil.wikipedia.org`.

AutoNav supplies developers with best-effort policies that aim to help them harden their websites. Using our parameterized simplification algorithm, developers get a slider style method for finding a trade-off between maintainability, performance and security. The simplification algorithm looks for evidence that all subdomains are trusted. The two sources used are the number of URLs that point to the subdomains (denoted t_1) and the number of subdomains that are pointed to (denoted t_2). The motivation for t_2 is that even if multiple links are found to `a.example.com` it does not imply that `b.example.com` should be allowed. Similarly, t_1 is motivated by the notion that the more URLs that point to `*.example.com`, the more it can be trusted. Figure A.4 shows the tree representation of 10 URLs pointing to `example.com` and its subdomains. u_i in the figure represents one URL, e.g. u_7 points to a resource on `test.b.www.example.com`. Furthermore, the figure also includes tuples of the threshold values (t_1, t_2). Figure A.5 shows the tree after simplification using a threshold of $(2, 2)$. Using this method the policy will only contain 3 entries instead of 7 entries.

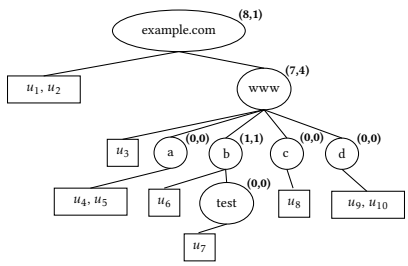


Figure A.4: Tree representation of 10 URLs collected from `example.com` and its subdomains. The tuples corresponds to the (t_1, t_2) thresholds.

Figure A.6 shows the result from crawling five pages on `ebay.com` and generating a policy. The crawler was only supplied with the start page and then found the other four using the crawling algorithm from Section A.5.3. The five pages crawled are shown in the middle of the figure in grey with integer labels. The arrows from these nodes indicate that a possible navigation was found between two nodes. The colors correspond to which part of the policy covers the navigation. As shown, `*.ebay.com` covers a lot of the subdomains, thus they all share the same color. Using the figure, an origin policy could be generated by taking the union of all the colors.

This method of generating policies guarantees that the functionality of the website will remain intact. This is because, if a domain is in the list of possible navigations, then it will be included in the policy. Similar to other policies, the generated policy would need to be recalculated if the website was updated to include new possible navigations. For security, the method guarantees that if a domain is not in the

list, then it will not be added to the policy. However, subdomains of domains in the list can be added to the policy.

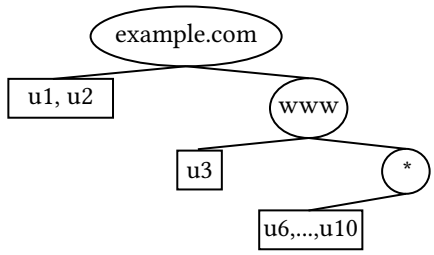


Figure A.5: Result of applying the simplification algorithm, using a treshold of $(2, 2)$, to the tree in Figure A.4. Resulting in the following policy, `navigate-to example.com www.example.com *.www.example.com`.

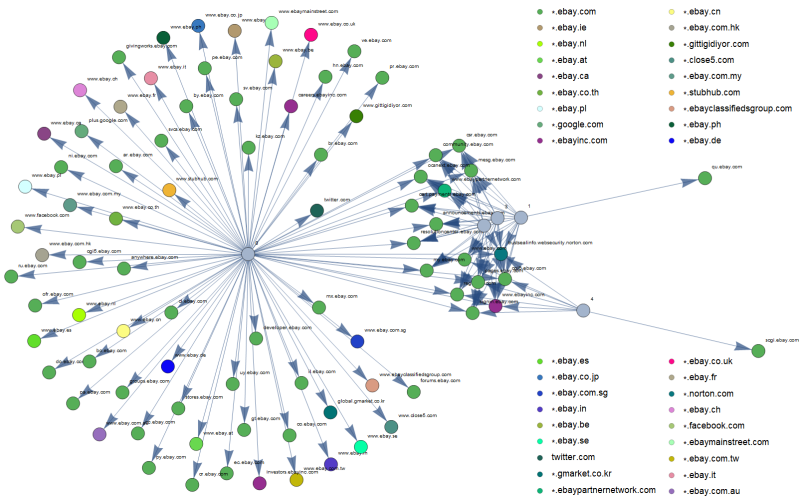


Figure A.6: Generated policies for `ebay.com`. The nodes with outwards pointing arrows are the five pages that we crawled. All the other nodes correspond to a possible navigation. The color indicates which part of the policy covers the navigation.

A.5.3 Crawling

Our implementation of AutoNav uses selenium with a Chrome instance to crawl the pages on a website. By only supplying AutoNav with the first URL it will automatically collect and crawl new URLs that it finds. When a URL from the same website is found it is added to a set of unvisited URLs, from which the next URL is picked. For

each page on a domain, all the JavaScript is executed, then the URLs from links and forms are saved. When the crawling session is over, the inference method described in Section A.5.1 is used to generate the policy.

A.5.4 Limitations

We did not take special care to crawl behind the login. However, it is trivial for a site owner to add a session cookie to the crawler. The more pages AutoNav can crawl the more the policy will cover. Crawling too few pages will result in an incomplete yet secure policy. The policy is secure because AutoNav will never add a domain to the policy that has not been seen.

We use static links to infer the policies, which will miss possible redirections. While not a security concern, we would produce more precise policies if each link was followed dynamically and the redirections recorded.

User-agent sniffing is a common problem for crawling studies. Since the AutoNav is designed for developers we think they can manually add entries such as `languages.mysite.org` and use the AutoNav to detect everything else.

A.6 Empirical Study

This section presents an empirical study to evaluate the performance impact of the new directive, as well as, how different delivery methods and simplifications can reduce the impact. Next, we evaluate AutoNav in how well automatically generated policies based on a subset of the website cover the full website.

To test how the new `navigate-to` policy will function on common websites we utilize AutoNav in a crawling experiment. For calculating the performance impact in Section A.6.1, we use Alexa's top 10,000 websites. For evaluating AutoNav itself we use Alexa's top 14,000, ensuring we have 10,000 domains which all have more than 100 pages each.

A.6.1 Policy tradeoffs

This section presents the performance tradeoffs between per-page and origin-wide policies together with the delivery methods of HTTP headers and origin policy.

The costs in Table A.3 are based on a user visiting n pages on a website, thus the cost of HTTP headers need aggregation over all pages, i.e. $\sum_{i \leq n}$. The cost of sending a single CSP policy depends on the number of URLs it contains. We defined the cost of the policy based on the set of URLs, i.e. $|U_i|$, U_i being the set of URLs on page i. Further, we can define a set of all URLs as the union of the sets of URLs on each page as $\bigcup_{j \leq n} U_j$, with corresponding

Empirical performance Based on the 10,000 crawled domains, a per-page policy, without any simplifications, would increase the header size with 215 bytes, per response. A more maintainable origin-wide policy results in a size increase to 1904 bytes. This cost can be decreased by using the origin policy for delivery, in which case the user only downloads the policy once. Note, as shown in Table A.3, that an

origin policy outperforms a per-page policy after only 9 responses. While per-page policies might seem better, they are difficult to use since they require knowledge about the content on each page. As such, some website, e.g. Facebook, use origin-wide policies, motivating the need for an origin policy delivery method.

In addition to the comparison between per-page and origin policy, we also evaluated the cost benefits of using our policy simplification algorithm. Using maximum simplifications, i.e. $t_1 = 1, t_2 = 1$, the average size of the origin wide policy decreases from 1904 to 1004 bytes, a decrease of 47%. Similarly, the per-page policy decreases from 215 bytes to 129 bytes, which is a 40% decrease. For some websites, the benefit of simplification is much greater. In particular, this is the case when websites allow navigation to numerous subdomains. For example, `spravker.ru` would require a 20438 byte origin policy without simplification, but only 61 bytes after simplification. The big difference stems from the fact that `spravker.ru` have 954 subdomains.

Table A.3: Empirical costs for different policy models.

	HTTP	**Origin Policy**
Per-page	$\sum_{i \leq n} 215$	-
Origin-wide	$\sum_{i \leq n} 1904$	1904

We also performed a more in-depth analysis of three websites, `ebay.com`, `wikipedia.org` and `stackexchange.com`, to see how the threshold affect performance. Fixing t_1 to 0, we only focus on the number of subdomains when deciding if wildcards should be used. Figure A.7 shows these domains as solid lines, together with the corresponding costs for their origin policies. As can be noted, after the t_2 threshold reaches 280 subdomains Wikipedia can no longer use the wildcard and the policy quickly increases in size. By increasing t_1 to 1000, more URLs are required before simplifications can take place. As can be seen in the dashed lines in Figure A.7, the crawled data from Wikipedia did not contain enough URLs to the same domain for a simplification. This would be the desired behavior if Wikipedia required high assurance before introducing wildcards.

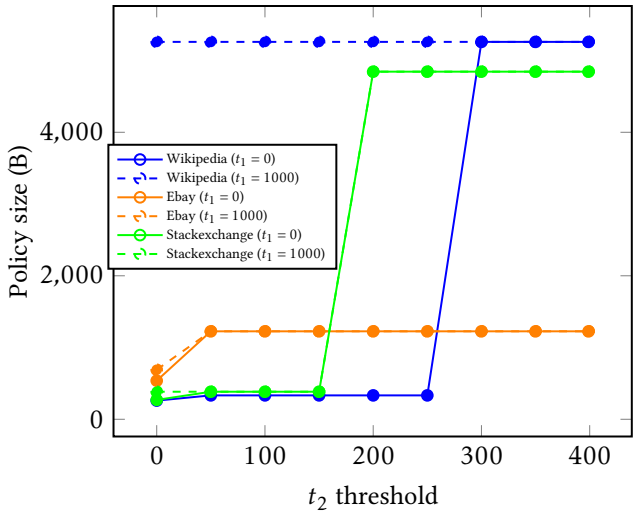


Figure A.7: Cost of origin policy for different domains and simplification thresholds. The y-axis shows policy size in bytes and the x-axis shows the t_2 threshold. The legend shows the t_1 threshold

A.6.2 Coverage

While full coverage may be desirable, the goal of AutoNav is to help even if the coverage is not complete, by providing a useful baseline policy for developers to build on.

Our coverage was generated similarly to the method used in CSPAutoGen [37]. We generate the policy based on a training set of 80% of the pages on a domain and then test how well they match the other 20%. We define U as the set of URLs in the training set. For the n pages in the validation set, we check if URLs on the page are covered, i.e. $p_i \subseteq U$, where p_i is the set of all the URLs on page p_i. Finally the coverage of a website is calculated as: $c = \frac{|\{p_i : p_i \subseteq U\}|}{n}$

Using this formula, c is calculated for all the websites that were crawled. In total 42% of all the websites were fully covered and for 59% of the websites 95% or more were covered. Note that these results come from only crawling 100 pages, deeper crawls can greatly increase this coverage.

A.7 Related work

Automatic methods for generating CSP policies have been studied before [12, 16, 18, 37]. deDacota by Doupe et al. [12] performs static analysis of ASP.NET code in order to separate JavaScript code from data. After the JavaScript has been separated into files, a CSP policy was generated for the file. AutoCSP by Fazzini et al. [16] takes a similar approach by analysing server-side code, PHP in this case. However, AutoCSP uses dynamic taint tracking instead of static analysis, allowing it to create policies

for inline JavaScript events and CSS code. While both AutoCSP and deDacota were successful, they required access to the source code of the application. In contrast, AutoNav uses a black-box approach which removes the need for the source code. Furthermore, the aforementioned methods focus on JavaScript and CSS, while our focus is on navigation and URLs. In addition, static analysis of source code will miss many URLs since modern web applications, like WordPress, store content in the database and not in the code.

In addition, research has been done on generating policies without access to the source code. Golubovic's autoCSP [18] method utilizes a reverse proxy and the report function in CSP to run an application in *learning mode*. In this mode, the tool externalizes inline code and generates policies for the scripts that should be allowed. A drawback is that autoCSP requires manual navigation through the application to ensure all scripts are triggered. While this works well for scripts, it becomes challenging when all possible links need to be navigated. A similar approach based on the report function in CSP was utilized by King's Firefox extension Laboratory [30]. Laboratory is impressive as it enables users to record and generate CSP policies in real-time while visiting a website. Starting with a strict policy, it gradually weakens it as violation reports are received. While this method could be extended to include navigations, it would require the user to initiate all possible navigations on each page. Instead of relying on the reporting functionality, our method uses a combination of static and dynamic analysis to record the navigations a document can initiate. By doing this we avoid the problem of having to initiate all navigations to generate a report. We also improve on the manual aspect of traversing a website by implementing an automatic crawler, as suggested by Golubovic, in future works.

CSPAutoGen Pan et al. [37] is also intended to automatize CSP generation. CSPAutoGen uses a crawler to analyze websites and try to infer which scripts should be allowed. Similar scripts are also generalized into abstract syntax trees, based on how many similar scripts are found. Once a policy has been inferred, CSPAutoGen functions as a proxy between the client and the server. This enables CSPAutoGen to rewrite requests and responses in real-time, without needing any CSP configurations on the website. This is a great feature when a server needs to be secured without any direct modification. While a similar approach could be used for URLs and navigation, our goal is to generate CSP policies that can be used by the server directly.

In addition to policy generation, we benefit from origin-wide policies [59]. Similarly to the work on evaluating general origin-wide policies by Van Acker et al. [50], our results also indicate that an origin-wide policy provides additional security without degrading performance.

A.8 Conclusion

Security We have performed a security analysis of the emerging CSP directive `navigate-to`. Our findings show that the current specification and implementations introduce new vulnerabilities. The vulnerabilities include methods for resource probing, login detection, circumventing blockage of third-party cookies, as

well as, history enumeration. To mitigate these problems we propose countermeasures to both the specification and implementation of the directive. We demonstrate that the directive triggers vulnerabilities even in websites that do not use the directive in their policies. Thus, we also propose countermeasures web developers can make to their applications in order to mitigate the possibilities of being exploited.

Automatization We have evaluated the possibility of automatically generating policies to help developers adopt the policy, we created AutoNav. AutoNav uses a black-box approach to crawl websites and generate CSP policies that can be directly applied to the website. Our results show that in total 42% of all the websites were fully covered and for 59% of the websites 95% or more were covered. We further simplify the process by identifying categories of websites which the policy better fits. Our research shows that shopping and adult websites are best covered. These websites have a high incentive to keep the users on their site, with the exception of linking to sponsors or partners, which AutoNav's policies cover.

Performance To analyze the performance of `navigate-to` we have conducted an empirical study of Alexa's top 10,000 websites. For each website, we have crawled 100 pages and based on theses generated policies. We show that on average this directive would increase the header size by 215 bytes per request. However, using our simplification algorithm we produce more maintainable policies which were also 40% smaller on average. Our results indicate that using an origin policy would require a one time cost of 1904 bytes, or 1004 using simplifications, as opposed to 215 bytes per request. Thus we show that the performance hit from the increased security can be efficiently mitigated by adopting an origin policy with suitable simplifications.

Coordinated disclosure We are in the process of disclosing the discovered vulnerabilities to the affected vendors, including Google where both Chrome's implementation of `navigate-to` directive and the Google Search website are affected. Based on our recommendations Firefox chose to harden their implementation against exfiltration attacks, as explained in Section A.4.2.2.

Acknowledgements Thanks are due to Mike West, Christoph Kerschbaumer, and Daniel Hausknecht for helpful discussions on the topic of `navigate-to`. This work was partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

Bibliography

[1] F. Aboukhadijeh. Is google an open redirector?, 2011.

[2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF)*, pages 290–304. IEEE, 2010.

[3] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 377–392, 2018.

[4] Apple, Google, Mozilla, Microsoft. Html living standard, 2019.

[5] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88. ACM, 2008.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Commun. ACM*, 52(6):83–91, 2009.

[7] M. A. Bashir, S. Arshad, W. Robertson, and C. Wilson. Tracing information flows between ad exchanges using retargeted ads. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 481–496, 2016.

[8] BuiltWith Pty Ltd. Prestashop usage statistics, 2018.

[9] S. Calzavara, A. Rabitti, and M. Bugliesi. Semantics-based analysis of content security policy deployment. *ACM Trans. Web*, 12(2), Jan. 2018.

[10] C. Castelluccia, E. De Cristofaro, and D. Perito. Private information disclosure from web searches. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 38–55. Springer, 2010.

[11] Dottoro. navigate method (window), 2019.

[12] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna. dedacota: toward preventing server-side xss via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 1205–1216, New York, NY, USA, 2013. ACM.

[13] A. Elsobky. Novel techniques for user deanonymization attacks, 2016.

[14] S. Englehardt and A. Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1388–1401. ACM, 2016.

[15] Facebook Inc. Use facebook pixel, 2018.

[16] M. Fazzini, P. Saxena, and A. Orso. Autocsp: Automatically retrofitting csp to web applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 336–346, May 2015.

[17] N. Gelernter and A. Herzberg. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1394–1405. ACM, 2015.

[18] N. Golubovic. autocsp - csp-injecting reverse http proxy, 2013.

[19] Google Inc. Personalized search for everyone, 2009.

[20] G. G. Gulyás, D. F. Somé, N. Bielova, and C. Castelluccia. To extend or not to extend: on the uniqueness of browser extensions and web logins. *CoRR*, abs/1808.07359, 2018.

[21] S. Helme. Optimising twitter's csp header, Jan 2018.

[22] J. Hodges, C. Jackson, and A. Barth. Http strict transport security (hsts). RFC 6797, RFC Editor, November 2012.

[23] E. Homakov. Using content-security-policy for evil, Jan 2014.

[24] D. Inc. Measuring the "filter bubble": How google is influencing what you click, 2018.

[25] C. Jackson and A. Barth. Forcehttps: protecting high-security web sites from network attacks. In *Proceedings of the 17th international conference on World Wide Web*, pages 525–534. ACM, 2008.

[26] J. Karahalis. Content security policy (csp), 2018.

[27] K. Karlsson. 179426 reflected xss on blockchain.info, 2017.

[28] C. Kerschbaumer. 1529068 - implement csp 'navigate-to' directive, February 2018.

[29] A. King. Allow navigation to only whitelisted urls via navigate-to 125, 2016.

[30] A. King. april/laboratory, 2018.

[31] D. Kristol and L. Montulli. Http state management mechanism. RFC 2965, RFC Editor, October 2000.

[32] R. Linus. Your social media fingerprint, 2017.

[33] J. Medley. Content-security-policy-report-only, 2018.

[34] J. Morrison. Open redirect urls: Is your site being abused?, 2009.

[35] A. Paicu. CSP 'navigate-to' directive: Consensus & Standardization, 2018.

[36] A. Paicu. Implement the 'navigation-to' directive, 2018.

[37] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 653–665, New York, NY, USA, 2016. ACM.

[38] P. Papadopoulos, N. Kourtellis, and E. Markatos. Cookie synchronization: Everything you always wanted to know but were afraid to ask. In *The World Wide Web Conference*, pages 1432–1442. ACM, 2019.

[39] P. D. Ryck, L. Desmet, F. Piessens, and M. Johns. *Primer on Client-Side Web Security*. Springer, 2014.

[40] G. B. Security. Communication with google's blink security team, November 2018.

[41] R. Singel. Google catches bing copying; microsoft says 'so what?, February 2011.

[42] N. Statt. Advertisers are furious with apple for new tracking restrictions in safari 11, 2017.

[43] M. Steffens, C. Rossow, M. Johns, and B. Stock. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. In *NDSS*, 2019.

[44] M. Stockley. Anatomy of a browser dilemma - how hsts supercookies make you choose between privacy or security, 2015.

[45] The OWASP Foundation. Owasp top 10 - 2017, 2017.

[46] The OWASP Foundation. Unvalidated redirects and forwards cheat sheet, 2017.

[47] The OWASP Foundation. Cross-site request forgery (csrf), 2018.

[48] The OWASP Foundation. Cross-site scripting (xss), 2018.

[49] S. Van Acker, D. Hausknecht, and A. Sabelfeld. Data exfiltration in the face of csp. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 853–864. ACM, 2016.

[50] S. Van Acker, D. Hausknecht, and A. Sabelfeld. Raising the bar: Evaluating origin-wide security manifests. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 342–354, 2018.

[51] A. van Kesteren. Fetch standard, February 2018.

[52] A. van Kesteren. Fetch living standard, 2019.

[53] M. Vasigh. Window.open(), 2018.

[54] G. Venkatadri, A. Andreou, Y. Liu, A. Mislove, K. P. Gummadi, P. Loiseau, and O. Goga. Privacy risks with facebook's pii-based targeting: Auditing a data broker's advertising interface. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 89–107, May 2018.

[55] T. Watanabe, E. Shioji, M. Akiyama, K. Sasaoka, T. Yagi, and T. Mori. User blocking considered harmful? an attacker-controllable side channel to identify social accounts. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 323–337. IEEE, 2018.

[56] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1376–1387. ACM, 2016.

[57] M. Weissbacher, T. Lauinger, and W. Robertson. Why is csp failing? trends and challenges in csp adoption. In *International Workshop on Recent Advances in Intrusion Detection*, pages 212–233. Springer, 2014.

[58] M. West. Allow navigation to only whitelisted urls via navigate-to 125, 2016.

[59] M. West. Origin policy, 2017.

[60] M. West. Origin policy, 2017.

[61] M. West. Content security policy level 3, 2018.

[62] M. West. Incrementally better cookies, May 2019.

[63] M. West, A. Barth, and D. Veditz. Content security policy level 2, 2016.

[64] Yan (bcrypt). @bcrypt - advanced browser fingerprinting - toorcon 2015, November 2015.

[65] M. Zalewski. Postcards from the post-xss world, 2011.

B

Black Widow: Blackbox Data-driven Web Scanning

Abstract. Modern web applications are an integral part of our digital lives. As we put more trust in web applications, the need for security increases. At the same time, detecting vulnerabilities in web applications has become increasingly hard, due to the complexity, dynamism, and reliance on third-party components. Blackbox vulnerability scanning is especially challenging because (i) for deep penetration of web applications scanners need to exercise such browsing behavior as user interaction and asynchrony, and (ii) for detection of non-trivial injection attacks, such as stored cross-site scripting (XSS), scanners need to discover inter-page data dependencies.
This paper illuminates key challenges for crawling and scanning the modern web. Based on these challenges we identify three core pillars for deep crawling and scanning: navigation modeling, traversing, and tracking inter-state dependencies. While prior efforts are largely limited to the separate pillars, we suggest an approach that leverages all three. We develop Black Widow, a blackbox data-driven approach to web crawling and scanning. We demonstrate the effectiveness of the crawling by code coverage improvements ranging from 63% to 280% compared to other crawlers across all applications. Further, we demonstrate the effectiveness of the web vulnerability scanning by featuring no false positives and finding more cross-site scripting vulnerabilities than previous methods. In older applications, used in previous research, we find vulnerabilities that the other methods miss. We also find new vulnerabilities in production software, including HotCRP, osCommerce, PrestaShop and WordPress.

B.1 Introduction

Ensuring the security of web applications is of paramount importance for our modern society. The dynamic nature of web applications, together with a plethora of different languages and frameworks, makes it particularly challenging for existing approaches to provide sufficient coverage of the existing threats. Even the web's main players, Google and Facebook, are prone to vulnerabilities, regularly discovered by security researchers. In 2019 alone, Google's bug bounty paid $6.5 million [16] and

Facebook $2.2 million [12], both continuing the ever-increasing trend. *Cross-Site Scripting (XSS)* attacks, injecting malicious scripts in vulnerable web pages, represent the lion's share of web insecurities. Despite mitigations by the current security practices, XSS remains a prevalent class of attacks on the web [38]. Google rewards millions of dollars for XSS vulnerability reports yearly [21], and XSS is presently the most rewarded bug on both HackerOne [20] and Bugcrowd [5]. This motivates the focus of this paper on detecting vulnerabilities in web applications, with particular emphasis on XSS.

Blackbox web scanning. When such artifacts as the source code, models describing the application behaviors, and code annotations are available, the tester can use whitebox techniques that look for vulnerable code patterns in the code or vulnerable behaviors in the models. Unfortunately, these artifacts are often unavailable in practice, rendering whitebox approaches ineffective in such cases.

The focus of this work is on *blackbox* vulnerability detection. In contrast to whitebox approaches, blackbox detection techniques rely on no prior knowledge about the behaviors of web applications. This is the standard for security penetration testing, which is a common method for finding security vulnerabilities [31]. Instead, they acquire such knowledge by interacting with running instances of web applications with *crawlers*. Crawlers are a crucial component of blackbox scanners that explore the attack surface of web applications by visiting webpages to discover URLs, HTML form fields, and other input fields. If a crawler fails to cover the attack surface sufficiently, then vulnerabilities may remain undetected, leaving web applications exposed to attacks.

Unfortunately, having crawlers able to discover in-depth behaviors of web applications is not sufficient to detect vulnerabilities. The detection of vulnerabilities often requires the generation of tests that can interact with the web application in non-trivial ways. For example, the detection of stored cross-site scripting vulnerabilities (stored XSS), a notoriously hard class of vulnerabilities [38], requires the ability to reason about the subtle dependencies between the control and data flows of web application to identify the page with input fields to inject the malicious XSS payload, and then the page that will reflect the injected payload.

Challenges. Over the past decade, the research community has proposed different approaches to increase the coverage of the attack surface of web applications. As JavaScript has rendered webpages dynamic and more complex, new ideas were proposed to incorporate these dynamic behaviors to ensure a correct exploration of the page behaviors (jÄk [30]) and the asynchronous HTTP requests (CrawlJAX [4, 26]). Similarly, other approaches proposed to tackle the complexity of the server-side program by reverse engineering (LigRE [10] and KameleonFuzz [11]) or inferring the state (Enemy of the State [8]) of the server, and then using the learned model to drive a crawler.

Unfortunately, despite the recent efforts, existing approaches do not offer sufficient coverage of the attack surface. To tackle this challenge, we start from two observations. First, while prior work provided solutions to individual challenges, leveraging their carefully designed combination has the potential to significantly improve the state of the art of modern web application scanning. Second, existing

solutions focus mostly on handling control flows of web applications, falling short of taking into account intertwined dependencies between control and data flows. Consider, for example, the dependency between a page to add new users and the page to show existing users, where the former changes the state of the latter. Being able to extract and use such an inter-page dependency will allow scanners to explore new behaviors and detect more sophisticated XSS vulnerabilities.

Contributions. This paper presents Black Widow, a novel blackbox web application scanning technique that identifies and builds on three pillars: navigation modeling, traversing, and tracking inter-state dependencies.

Given a URL, our scanner creates a navigation model of the web application with a novel JavaScript dynamic analysis-based crawler able to explore both the static structure of webpages, i.e., anchors, forms, and frames, as well as discover and fire JavaScript events such as mouse clicks. Also, our scanner further annotates the model to capture the sequence of steps required to reach a given page, enabling the crawler to retrace its steps. When visiting a webpage, our scanner enriches our model with data flow information using a black-box, end-to-end, dynamic taint tracking technique. Here, our scanner identifies input fields, i.e., taint source, and then probe them with unique strings, i.e., taint values. Later, the scanner checks when the strings re-surface in the HTML document, i.e., sinks. Tracking these taints allows us to understand the dependencies between different pages.

We implement our approach as a scanner on top of a modern browser with a state-of-the-art JavaScript engine. To empirically evaluate it, both in terms of coverage and vulnerability detection, we test it on two sets of web applications and compare the results with other scanners. The first set of web applications are older well-known applications that have been used for vulnerability testing before, e.g. WackoPicko and SCARF. The second set contains new production applications such as CMS platforms including WordPress and E-commerce platforms including PrestaShop and osCommerce. From this, we see that our approach improves code coverage by between 63% and 280% compared to other scanners across all applications. Across all web applications, our approach improves code coverage by between 6% and 62%, compared to the *sum* of all other scanners. In addition, our approach finds more XSS vulnerabilities in older applications, i.e. phpBB, SCARF, Vanilla and WackoPicko, that have been used in previous research. Finally, we also find multiple new vulnerabilities across production software including HotCRP, osCommerce, PrestaShop and WordPress.

Finally, while most scanners produce false positives, Black Widow is free of false positives on the tested applications thanks to its dynamic verification of code injections.

In summary, the paper offers the following contributions.

- We identify unsolved challenges for scanners in modern web applications and present them in Section B.2.
- We present our novel approaches for finding XSS vulnerabilities using inter-state dependency analysis and crawling complex workflows in Section C.2.

- We implement and share the source code of Black Widow[1]
- We perform a comparative evaluation of Black Widow on 10 popular web applications against 7 web application scanners.
- We present our evaluation in Section B.4 showing that our approach finds 25 vulnerabilities, of which 6 are previously unknown in HotCRP, osCommerce, PrestaShop and WordPress. Additionally, we find more vulnerabilities in older applications compared to other scanners. We also improve code coverage on average by 23%.
- We analyze the results and explain the important features required by web scanners in Section B.5.

B.2 Challenges

Existing web application scanners suffer from a number of shortcomings affecting their ability to cope with the complexity of modern web applications [3, 9]. We observe that state-of-the-art scanners tend to focus on separate challenges to improve their effectiveness. For example, jÄk focuses on JavaScript events, Enemy of the State on application states, LigRE on reverse engineering and CrawlJAX on network requests. However, to successfully scan applications our insight is that these challenges must be solved simultaneously. This section focuses on these shortcomings and extracts the key challenges to achieve high code coverage and effective vulnerability detection.

High code coverage is crucial for finding any type of vulnerability as the scanner must be able to reach the code to test it. For vulnerability detection, we focus on stored XSS as it is known to be difficult to detect and a category of vulnerabilities poorly covered by existing scanners [3, 9]. Here the server stores and uses at a later time untrusted inputs in server operations, without doing proper validation of the inputs or sanitization of output.

A web application scanner tasked with the detection of subtle vulnerabilities like stored XSS faces three major challenges. First, the scanner needs to model the various states forming a web application, the connections and dependencies between states (Section B.2.1). Second, the identification of these dependencies requires the scanner to be able to traverse the complex workflows in applications (Section B.2.2). Finally, the scanner needs to track subtle dependencies between states of the web application (Section B.2.3).

B.2.1 Navigation Modeling

Modern web applications are dynamic applications with an abundance of JavaScript code, client-side events and server-side statefulness. Modeling the scanner's interaction with both server-side and client-side code is complicated and challenging. Network requests can change the state of the server while clicking a button can result in changes to the DOM, which in turn generates new links or fields. These

[1]Our implementation is available online on `https://www.cse.chalmers.se/research/group/security/black-widow/`

orthogonal problems must all be handled by the scanner to achieve high coverage and improved detection rate of vulnerabilities. Consider the flow in an example web application in Figure B.1. The scanner must be able to model links, forms, events and the interaction between them. Additionally, to enable workflow traversal, it must also model the path taken through the application. Finally, the model must support inter-state dependencies as shown by the dashed line in the figure.

The state-of-the-art consists of different approaches to navigation modeling. Enemy of the State uses a state machine and a directed graph to infer the server-side state. However, the navigation model lacks information about client-side events. In contrast, jÄk used a graph with lists inside nodes, to represent JavaScript events. CrawlJAX moved the focus to model JavaScript network requests. While these two model client-side, they miss other important navigation methods such as form submissions.

A navigation model should allow the scanner to efficiently and exhaustively scan a web application. Without correct modeling, the scanner will miss important resources or spend too much time revisiting the same or similar resources. To achieve this, the model must cover a multitude of methods for interaction with the application, including GET and POST requests, JavaScript events, HTML form and iframes.

In addition, the model should be able to accommodate dependencies. Client-side navigations, such as clicking a button, might depend on previous events. For example, the user might have to hover the menu before being able to click the button. Similarly, installation wizards can require a set of forms to be submitted in sequence.

With a solution to the modeling challenge, the next challenge is how the scanner should use this model, i.e. how should it traverse the model.

B.2.2 Traversing

To improve code coverage and vulnerability detection, the crawler component of the scanner must be able to traverse the application. In particular, the challenge of reproducing workflows is crucial for both coverage and vulnerability detection. The challenges of handling complex workflows include deciding in which order actions should be performed and when to perform possibly state-changing actions, e.g. submitting forms. Also, the workflows must be modeled at a higher level than network requests as simply replaying requests can result in incorrect parameter values, especially for context-dependent value such as a comment ID. In Figure B.1, we can observe a workflow requiring a combination of normal link navigation, form submission and event interaction. Also, note that the forms can contain security nonces to protect against CSRF attacks. A side effect of this is that the scanner can not replay the request and just change the payload, but has to reload the page and resubmit the form.

The current state-of-the-art focuses largely on navigation and exploration but misses out on global workflows. Both CrawlJAX and jÄk focused on exploring client-side events. By exploring the events in a depth-first fashion, jÄk can find sequences of events that could be exploited. However, these sequences do not extend across multiple pages, which will miss out on flows. Enemy of the State takes the opposite approach and ignores traversing client-side events and instead focuses on

traversing server-side states. To traverse, they use a combination of picking links from the previous response and a heuristic method to traverse edges that are the least likely to result in a state change, e.g. by avoiding form submission until necessary. To change state they sometimes need to replay the request from the start. Replaying requests may not be sufficient as a form used to post comments might contain a submission ID or view-state information that changes for each request. Due to the challenge of reproducing these flows, their approach assumes the power to reset the full application when needed, preventing the approach from being used on live applications.

We note that no scanner handles combinations of events and classic page navigations. Both jÄk and CrawlJAX traverse with a focus on client-side state while Enemy of the State focus on links and forms for interaction. Simply combining the two approaches of jÄk and Enemy of the State is not trivial as their approaches are tailored to their goals. Enemy of the State uses links on pages to determine state changes, which are not necessarily generated by events.

Keeping the scanner authenticated is also a challenge. Some scanners require user-supplied patterns to detect authentication [28, 34, 36]. jÄk authenticates once and then assumes the state is kept, while CrawlJAX ignores it altogether. Enemy of the State can re-authenticate if they correctly detect the state change when logging out. Once again it is hard to find consensus on how to handle authentication.

In addition to coverage, traversing is important for the fuzzing part of the scanner. Simply exporting all requests to a standalone fuzzer is problematic as it results in loss of context. As such, the scanner must place the application in an appropriate state before fuzzing. Here some scanners take the rather extreme approach of trying to reset the entire web application before fuzzing each parameter [8, 10, 11]. jÄk creates a special attacker module that loads a URL and then executes the necessary events. This shows that in order to fuzz the application in a correct setting, without requiring a full restart of the application, the scanner must be able to traverse and attack both server-side and client-side components.

Solving both modeling and traversing should enable the scanner to crawl the application with improved coverage, allowing it to find more parameters to test. The final challenge, particularly with respect to stored XSS, is mapping the dependencies between different states in the application.

B.2.3 Inter-state Dependencies

It is evident that agreeing on a model that fits both client-side and server-side is hard, yet important. In addition, neither of the previous approaches are capable of modeling inter-state dependencies or general workflows. While Enemy of the State model states, they miss the complex workflows and the inter-state dependencies. The model jÄk uses can detect workflows on pages but fails to scale for the full application.

A key challenge faced by scanners is how to accurately and precisely model how user inputs affect web applications. As an example, consider the web application workflow in Figure B.1 capturing an administrator registering a new user. In this workflow, the administrator starts from the index page (i.e., `index.php`) and navi-

gates to the login page (i.e., `login.php`). Then, the administrator submits the password and lands on the administrator dashboard (i.e., `admin.php`). From the dashboard, the administrator reaches the user management page (i.e., `admin.php#users`), and submits the form to register a new user. Then, the web application stores the new user data in the database, and, as a result of that, the data of the new user is shown when visiting the page of existing users (i.e., `view_users.php`). Such a workflow shows two intricate dependencies between two states of the web application: First, an action of `admin.php#users` can cause a transition of `view_users.php`, and second, the form data submitted to `admin.php#users` is reflected in the new state of `admin.php#users`.

To detect if the input fields of the form data are vulnerable to, e.g., cross-site scripting (XSS), a scanner needs to inject payloads in the form of `admin.php#users` and then reach `view_users.php` to verify whether the injection was successful. Unfortunately, existing web scanners are not aware of these inter-state dependencies, and after injecting payloads, they can hardly identify the page where and whether the injection is reflected.

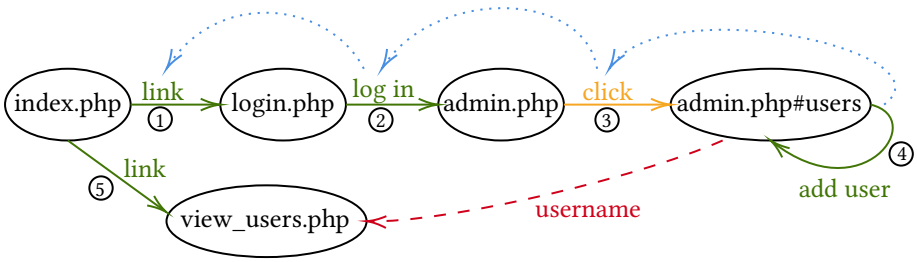


Figure B.1: Example of a web application where anyone can see the list of users and the admin can add new users. The dashed red line represents the inter-state dependency. Green lines are HTML5 and orange symbolises JavaScript. The dotted blue lines between edges would be added by our scanner to track its path. The sequence numbers shown the necessary order to find the inter-state dependency.

B.3 Approach

Motivated by the challenges in Section B.2, this section presents our approach to web application scanning. The three key ingredients of our approach are edge-driven navigation with path-augmentation, complex workflow traversal, and fine-grained inter-state dependency tracking. We explain how we connect these three parts in Figure C.11. In addition to the three main pillars, we also include a section about the dynamic XSS detection used in Black Widow and motivate why false positives are improbable.

Figure C.11 takes a single target URL as an input. We start by creating an empty node, allowing us to create an initial edge between the empty node and the node

containing the input URL. The main loop picks an unvisited edge from the navigation graph and then traverses it, executing the necessary workflows as shown in Figure B.3. In Figure B.3, we use the fact that each edge knows the previous edge. The `isSafe` function in Figure B.3 checks if the type of action, e.g. JavaScript event or form submission, is *safe*. We consider a type to be *safe* if it is a GET request, more about this in Section B.3.2. Once the safe edge is found we navigate the chain of actions. Following this navigation, the scanner is ready to parse the page. First, we inspect the page for inter-state dependency tokens and add the necessary dependency edges, as shown in Figure B.4. Each `token` will contain a taint value, explained more in Section B.3.3, a source edge and a sink edge. If a source and sink are found, our scanner will fuzz the source and check the sink. Afterward, we extract any new possible navigation resources and add them to the graph. Next, we fuzz any possible parameters in the edge and then inject a taint token. The order is important as we want the token to overwrite any stored fuzzing value. Finally, the edge is marked as visited and the loop repeats.

The goal of this combination is to improve both vulnerability detection and code coverage. The three parts of the approach support each other to achieve this. A strong model that handles different navigation methods and supports augmentation with path and dependency information will enable a richer interaction with the application. Based on the model we can build a strong crawler component that can handle complex workflow which combines requests and client-side events. Finally, by tracking inter-state dependencies we can improve detection of stored vulnerabilities.

```
Data: Target url
Global: tokens // Used in Algorithm 3
Graph navigation; // Augmented navigation graph
navigation.addNode(empty);
navigation.addNode(url);
navigation.addEdge(empty, url);
while unvisited edge e in navigation do
    traverse(e); // See Algorithm 2
    inspectTokens(e, navigation); // See Algorithm 3
    resources = extract({urls, forms, events, iframes});
    for resource in resources do
        navigation.addNode(resource)
        navigation.addEdge(e.targetNode, resource)
    end
    attack(e);
    injectTokens(e);
    mark e as visited;
end
```

Figure B.2: Scanner algorithm

```
Function traverse(e: edge)
    workflow = []; // List of edges
    currentEdge = e;
    while prevEdge = currentEdge.previous do
        workflow.prepend(currentEdge);
        if isSafe(currentEdge.type) then
            break;
        end
        currentEdge = prevEdge
    end
    navigate(workflow);
end
```

Figure B.3: Traversal algorithm

```
Function inspectTokens(e: edge, g: graph)
    for token in tokens do
        if pageSource(e) contains token.value then
            token.sink = e;
            g.dependency(token.source, token.sink);
            attack(token.source, token.sink);
        end
    end
end
Function injectTokens(e: edge)
    for parameter in e do
        token.value = generateToken();
        token.source = e;
        tokens.append(token);
        inject token in parameter;
    end
end
```

Figure B.4: Inter-state dependency algorithms

B.3.1 Navigation Modeling

Our approach is model-based in the sense that it creates, maintains, and uses a model of the web application to drive the exploration and detection of vulnerabilities. Our model covers both server-side and client-side aspects of the application. The model tracks server-side inter-state dependencies and workflows. In addition, it directly captures elements of the client-side program of the web application, i.e., HTML and the state of the JavaScript program.

Model Construction Our model is created and updated at run-time while scanning the web application. Starting from an initial URL, our scanner retrieves the first webpage and the referenced resources. While executing the loaded JavaScript, it extracts the registered JavaScript events and adds them to our model. Firing an event may result in changing the internal state of the JavaScript program, or retrieving a new page. Our model captures all these aspects and it keeps track of the sequence of fired events when revisiting the web application, e.g., for the detection of vulnerabilities.

Accordingly, we represent web applications with a labeled directed graph, where each node is a state of the client-side program and edges are the action (e.g., click) to move from one state to another one. The state of our model contains both the state of the page, i.e., the URL of the page, and the state of the JavaScript program, i.e., the JavaScript event that triggered the execution. Then, we use labeled edges for state transitions. Our model supports four types of actions, i.e., GET requests, form submission, iframes and JavaScript events. While form submissions normally result in GET or POST requests, we need a higher-level model for the traversing method explained in Section B.3.2. We consider iframes as actions because we need to model the inter-document communication between the iframe and the parent, e.g firing an event in the parent might affect the iframe. By simply considering the iframe source as a separate URL, scanners will miss this interaction. Finally, we annotate each edge with the previous edge visited when crawling the web application, as shown in Figure B.1. Such an annotation will allow the crawler to reconstruct the *paths* within the web application, useful information for achieving deeper crawling and when visiting the web application for testing.

Extraction of Actions The correct creation of the model requires the ability to extract the set of possible actions from a web page. Our approach uses dynamic analysis approach, where we load a page and execute it in a modified browser environment, and then we observe the execution of the page, monitoring for calls to browser APIs to register JavaScript events and modification of the DOM tree to insert tags such as forms and anchors.

Event Registration Hooking Before loading a page we inject JavaScript which allows us to wrap functions such as `addEventListener` and detect DOM objects with event handlers. We accomplish this by leveraging the JavaScript libraries developed for the jÄk scanner [30]. While lightweight and easy to use, in-browser instrumentation is relatively fragile. A more robust approach could be directly modifying the JavaScript engine or source-to-source compile the code for better analysis.

DOM Modification To detect updates to the page we rescan the page whenever we execute an event. This allows us to detect dynamically added items.

Infinite Crawls When visiting a webpage, crawlers can enter in an infinite loop where they can perform the same operation endlessly. Consider the problem of crawling an online calendar. When a crawler clicks on the *View next week* button, the new page may have a different URL and content. The new page will container again the button *View next week*, triggering an infinite loop. An effective strategy to avoid infinite crawls is to define (i) a set of heuristics that determine when two pages or

two actions are similar, and (ii) a hard limit to the maximum number of "similar" actions performed by the crawler. In our approach, we define two pages to be similar if they share the same URL except for the query string and the fragments. For example, `https://example.domain/path/?x=1` and `https://example.domain/path/?x=2` are similar whereas `https://example.domain/?x=1` is different from this URL with another path `https://example.domain/path/?x=2`. The hard limit is a configuration parameter of our approach.

B.3.2 Traversal

To traverse the navigation model we pick unvisited edges from the graph in the order they were added, akin to breadth-first search. This allows the scanner to gain an overview of the application before diving into specific components. The edges are weighted with a positive bias towards form submission, which enables this type of deep-dive when forms are detected.

To handle the challenge of session management, we pay extra attention to forms containing password fields, as this symbolizes an opportunity to authenticate. Not only does this enable the scanner to re-authenticate but it also helps when the application generates a login form due to incorrect session tokens. Another benefit is a more robust approach to complicated login flows, such as double login to reach the administrator page—we observed such workflow in phpBB, one of the web applications that we evaluated.

The main challenge to overcome is that areas of a web application might require the user to complete a specific sequence of actions. This could, for example, be to review a comment after submitting it or submit a sequence of forms in a configuration wizard. It is also common for client-side code to require chaining, e.g. hover a menu before seeing all the links or click a button to dynamically generate a new form.

We devise a mechanism to handle navigation dependencies by modeling the workflows in the application. Whenever we need to follow an edge in the navigation graph, we first check if the previous edge is considered *safe*. Here we define safe to be an edge which represents a GET request, similar to the HTTP RFC [14]. If the edge is safe, we execute it immediately. Otherwise, we recursively inspect the previous edge until a safe edge is found, as shown in Figure B.3. Note that the first edge added to the navigation graph is always a GET request, which ensures a base case. Once the safe edge is found, we execute the full workflow of edges leading up to the desired edge. Although the RFC defines GET requests to be idempotent, developers can still implement state-changing functions on GET requests. Therefore, considering GET requests as *safe* is a performance trade-off. This could be deactivated by a parameter in Black Widow, causing the scanner to traverse back to the beginning.

Using Figure B.1 as an example if the crawler needed to submit a form on the page `admin.php#users` then it would first have to load `login.php` and then submit that form, followed by executing a JavaScript event to dynamically add the user form.

We chose to only chain actions to the previous GET request, as they are deemed safe. Chaining from the start is possible, but it would be slow in practice.

B.3.3 Inter-state Dependencies

One of the innovative aspects of our approach is to identify and map the ways user inputs are connected to the states of a web application. We achieve that by using a dynamic, end-to-end taint tracking while visiting the web application. Whenever our scanner identifies an input field, i.e., a *source*, it will submit a unique token. After that, the scanner will look for the token when visiting other webpages, i.e., *sinks*.

Tokens To map source and sinks, we use string tokens. We designed tokens to avoid triggering filtering functions or data validation checks. At the same time, we need tokens with a sufficiently high entropy to not be mistaken for other strings in the application. Accordingly, we generate tokens as pseudo-random strings of eight lowercase characters e.g. `frcvwwzm`. This is what `generateToken()` does in Figure B.4. This could potentially be improved by making the tokens context-sensitive, e.g. by generating numeric tokens or emails. However, if the input is validated to only accept numbers, for example, then XSS is not possible.

Sources and Sinks The point in the application where the token is injected defines the *source*. More specifically, the source is defined as a tuple containing the edge in the navigation graph and the exact parameter where the token was injected. The resource in the web application where the token reappears defines the *sink*. All the sinks matching a certain source will be added to a set which in turn is connected to the source. Similar to the sources, each sink is technically an edge since they carry more context than a resource node. Since each source can be connected to multiple sinks, the scanner needs to check each sink for vulnerabilities whenever a payload is injected into a source.

In our example in Figure B.1, we have one source and one connected sink. The source is the *username* parameter in the form on the management page and the sink is the view users page. If more parameters, e.g. email or signature, were also reflected then these would create new dependency edges in the graph.

B.3.4 Dynamic XSS detection

After a payload has been sent, the scanner must be able to detect if the payload code is executed. Black Widow uses a fine-grained dynamic detection mechanism, making false positives very improbable. We achieve this by injecting our JavaScript function `xss(ID)` on every page. This function adds `ID` to an array that our scanner can read. Every payload generated by Black Widow will try to call this function with a random ID, e.g. `<script>xss(71942203)</script>` Finally, by inspecting the array we can detect exactly which payloads resulted in code execution.

For this to result in a false positive, the web application would have to actively listen for a payload, extract the ID, and then run our injected `xss(ID)` function with a correct ID.

B.4 Evaluation

In this section, we present the evaluation of our approach and the results from our experiments. In the next section, we perform an in-depth analysis of the factors behind the results.

To evaluate the effectiveness of our approach we implement it in our scanner Black Widow and compare it with 7 other scanners on a set of 10 different web applications. We want to compare both the crawling capabilities and vulnerability detection capabilities of the scanners. We present the implementation details in Section B.4.1. The details of the experimental setup are presented in Section B.4.2. To measure the crawling capabilities of the scanners we record the code coverage on each of application. The code coverage results are presented in Section B.4.3. For the vulnerability detection capabilities, we collect the reports from each scanner. We present both the reported vulnerabilities and the manually verified ones in Section B.4.4.

Table B.1: Lines of code (LoC) executed on the server. Each column represents the comparison between Black Widow and another crawler. The cells contain three numbers: unique LoC covered by Black Widow ($A \setminus B$), LoC covered by both crawlers ($A \cap B$) and unique LoC covered by the other crawler ($B \setminus A$). The numbers in bold highlight which crawler has the best coverage.

Crawler	Arachni			Enemy			jÄk			Skipfish			w3af			Wget			ZAP		
	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$
Drupal	**35 146**	22 870	757	6 365	51 651	**20 519**	**25 198**	32 818	5 846	**29 873**	28 143	937	**32 213**	25 803	725	**32 981**	25 035	498	**15 610**	42 406	2 591
HotCRP	**2 416**	16 076	948	**16 573**	1 919	0	**6 771**	11 721	271	**11 295**	7 197	31	**3 217**	15 275	768	**16 345**	2 147	3	**16 001**	2 491	24
Joomla	**14 573**	29 263	1 390	**33 335**	10 501	621	**24 728**	19 108	1 079	**33 254**	10 582	328	**12 533**	31 303	1 255	**33 975**	9 861	576	**7 655**	36 181	1 659
osCommerce	**3 919**	6 722	172	**9 626**	1 015	15	**4 171**	6 470	507	**4 964**	5 677	110	**5 601**	5 040	661	**6 070**	4 571	103	**6 722**	3 919	209
phpBB	**2 822**	5 178	492	**2 963**	5 037	337	**3 150**	4 850	348	**4 643**	3 357	72	**4 312**	3 688	79	**4 431**	3 569	21	**4 247**	3 753	65
PrestaShop	**105 974**	75 924	65 650	**157 095**	24 803	3 332	**155 579**	26 319	58	**138 732**	43 166	1 018	**156 513**	25 385	3 053	**148 868**	33 030	118	**141 032**	40 866	110
SCARF	**189**	433	12	**270**	352	5	**342**	280	2	**464**	158	5	**404**	218	6	**520**	102	2	**340**	282	2
Vanilla	**5 381**	9 908	491	**6 032**	9 257	185	**3 122**	12 167	536	**8 285**	7 004	577	**8 202**	7 087	171	**8 976**	6 313	18	**8 396**	6 893	145
WackoPicko	**202**	566	2	**58**	710	9	**463**	305	0	**274**	494	14	**111**	657	9	**495**	273	0	**379**	389	2
WordPress	**8 871**	45 345	1 615	**35 092**	19 124	256	**18 572**	35 644	579	**7 307**	46 909	5 114	**26 785**	27 431	640	**37 073**	17 143	73	**25 732**	28 484	781

B.4.1 Implementation

Our prototype implementation follows the approach presented above in Section C.2. It exercises full dynamic execution capabilities to handle such dynamic features of modern applications like AJAX and dynamic code execution, e.g. `eval`. To achieve this we use Python and Selenium to control a mainstream web browser (Chrome). This gives us access to a state-of-the-art JavaScript engine. In addition, by using a mainstream browser we can be more certain that the web application is rendered as intended.

We leverage the JavaScript libraries developed for the jÄk scanner [30]. These libraries are executed before loading the page. This allows us to wrap functions such as `addEventListener` and detect DOM objects with event handlers.

B.4.2 Experimental Setup

In this section, we present the configuration and methodology of our experiments.

Code Coverage To evaluate the coverage of the scanners we chose to compare the lines of code that were executed on the server during the session. This is different from previous studies [8, 30], which relied on requested URLs to determine coverage. While comparing URLs is easier, as it does not require the web server to run in debug mode, deriving coverage from it becomes harder. URLs can contain random parameter data, like CSRF tokens, that are updated throughout the scan. In this case, the parameter data has a low impact on the true coverage. Conversely, the difference in coverage between `main.php?page=news` and `main.php?page=login` can be large. By focusing on the execution of lines of code we get a more precise understanding of the coverage.

Calculating the total number of lines of code accurately in an application is a difficult task. This is especially the case in languages like PHP where code can be dynamically generated server-side. Even if possible, it would not give a good measure for comparison as much of the code could be unreachable. This is typically the case for applications that have installation code, which is not used after completing it.

Instead of analyzing the fraction of code executed in the web application, we compare the number of lines of code executed by the scanners. This gives a relative measure of performance between the scanners. It also allows us to determine exactly which lines are found by multiple scanners and which lines are uniquely executed.

To evaluate the code coverage we used the Xdebug [33] module in PHP. This module returns detailed data on the lines of code that are executed in the application. Each request to the application results in a separate list of lines of code executed for the specific request.

Vulnerabilities In addition to code coverage, we also evaluate how good the scanners are at finding vulnerabilities. This includes how many vulnerabilities they can find and how many false positives they generate. While there are many vulnerability types, our study focuses on both reflected and stored XSS.

To evaluate the vulnerability detection capabilities of the scanners, we collect and process all the vulnerabilities they report. First, we manually analyze if the vulnerabilities can be reproduced or if they should be considered false positives. Second, we cluster similar vulnerability reports into a set of unique vulnerabilities to make a fair comparison between the different reporting mechanisms in the scanners. We do this because some applications, e.g. SCARF, can generate an infinite number of vulnerabilities by dynamically adding new input fields. These should be clustered together. Classifying the uniqueness of vulnerabilities is no easy task. What we aim to achieve is a clustering in which each injection corresponds to a unique line of code on the server. That is, if a form has multiple fields that are all stored using the same SQL query then all these should count as one injection. The rationale is that it would only require the developer to change one line in the server code. Similarly, for reflected injections, we cluster parameters of the same request together. We

manually inspect the web application source code for each reported true-positive vulnerability to determine if they should be clustered.

Scanners We compare our scanner Black Widow with both Wget [27] for code coverage reference and 6 state-of-the-art open-source web vulnerability scanners from both academia and the web security community: Arachni [36], Enemy of the State [8], jÄk [30], Skipfish [42], w3af [34] and ZAP [28]. We use Enemy of the State and jÄk as they are state-of-the-art academic blackbox scanners. Skipfish, Wget and w3af are included as they serve as good benchmarks when comparing with previous studies [8, 30]. Arachni and ZAP are both modern open-source scanners that have been used in more recent studies [19]. Including a pure crawler with JavaScript capabilities, such as CrawlJAX [26], could serve as a good coverage reference. However, in this paper we focus on coverage compared to other vulnerability scanners. We still include Wget for comparison with previous studies. While it would be interesting to compare our results with commercial scanners, e.g. Burp Scanner [32], the closed source nature of these tools would make any type of feature attribute hard.

We configure the scanners with the correct credentials for the web application. When this is not possible we change the default credentials of the application to match the scanner's default values. Since the scanners have different capabilities, we try to configure them with as similar configurations as possible. This entails activating crawling components, both static and dynamic, and all detection of all types of XSS vulnerabilities.

Comparing the time performance between scanners is non-trivial to do fairly as they are written in different languages and some are sequential while others run in parallel. Also, we need to run some older ones in VMs for compatibility reasons. To avoid infinite scans, we limit each scanner to run for a maximum of eight hours.

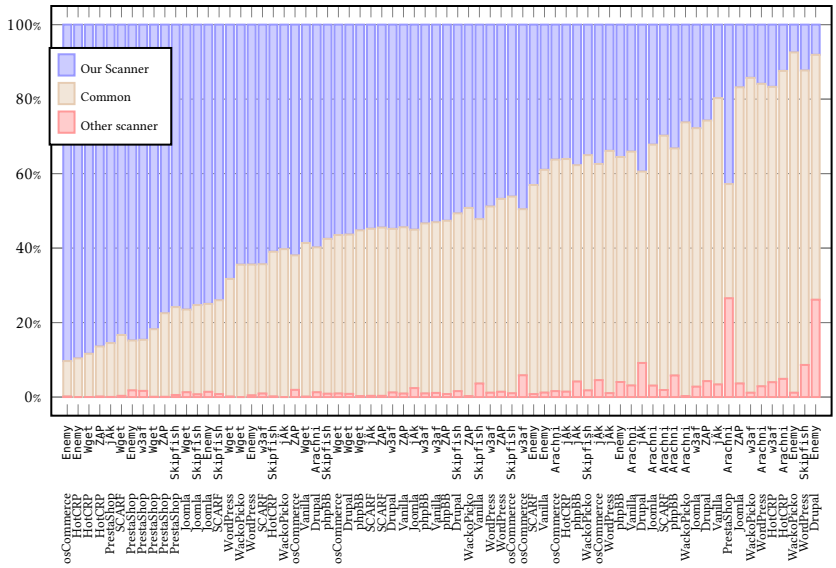


Figure B.5: Each bar compares our scanner to one other scanner on a web application. The bars show three fractions: unique lines we find, lines both find and lines uniquely found by the other scanner.

Web Applications To ensure that the scanners can handle different types of web applications we test them on 10 different applications. The applications range from reference applications that have been used in previous studies to newer production-grade applications. Each application runs in a VM that we can reset between runs to improve consistency.

We divide the applications into two different sets. Reference applications with known vulnerabilities: phpBB (2.0.23), SCARF (2007), Vanilla (2.0.17.10) and WackoPicko (2018); and modern production-grade applications: Drupal (8.6.15), HotCRP (2.102), Joomla (3.9.6), osCommerce (2.3.4.1), PrestaShop (1.7.5.1) and WordPress (5.1).

B.4.3 Code Coverage Results

This section presents the code coverage in each web application by all of the crawlers. Table B.1 shows the number of unique lines of code that were executed on the server. Black Widow has the highest coverage on 9 out of the 10 web applications.

Using Wget as a baseline Table B.1 illustrates that Black Widow increases the coverage by almost 500% in SCARF. Similarly with modern production software, like PrestaShop, we can see an increase of 256% in coverage compared to Wget. Even when comparing to state-of-the-art crawlers like jÄk and Enemy of the State we have more than 100% increase on SCARF and 320% on modern applications like PrestaShop. There is, however, a case where Enemy of the State has the highest coverage on Drupal. This case is discussed in more detail in Section B.5.1.

Table B.2: Unique lines our scanner finds ($A \setminus U$) compared to the union of all other scanners (U).

Application	Our scanner $A \setminus U$	Other scanners U	Improvement $\lvert A \setminus U \rvert / \lvert U \rvert$
Drupal	4 378	80 213	+5.5%
HotCRP	1 597	18 326	+8.7%
Joomla	5 134	42 443	+12.1%
osCommerce	2 624	9 216	+28.5%
phpBB	2 743	5 877	+46.7%
PrestaShop	95 139	153 452	+62.0%
SCARF	176	464	+37.9%
Vanilla	2 626	14 234	+18.4%
WackoPicko	50	742	+6.7%
WordPress	3 591	58 131	+6.2%

While it would be beneficial to know how far we are from perfect coverage, we avoid calculating a ground truth on the total number of lines of code for the applications as it is difficult to do in a meaningful way. Simply aggregating the number of lines in the source code will misrepresent dynamic code, e.g. `eval`, and count dead code, e.g. installation code.

We also compare Black Widow to the combined efforts of the other scanners to better understand how we improve the state-of-the-art. Table B.2 has three columns containing the number of lines of code that Black Widow finds which none of the others find, the combined coverage of the others and finally our improvement in coverage. In large applications, like PrestaShop, Black Widow was able to find 53 266 lines of code that none of the others found. For smaller applications, like phpBB, we see an improvement of up to 46.7% compared to the current state-of-the-art.

To get a better understanding of which parts of the application the scanners are exploring, we further compare the overlap in the lines of code between the scanners. In Table B.3 we present the number of unique lines of code Black Widow find compared to another crawler. The improvement is calculated as the number of unique lines we find divided by the *total* number of lines the other crawlers find.

We plot the comparison for all scanners on all platforms in Figure B.5. In this figure, each bar represents the fraction of lines of code attributed to each crawler. At the bottom is the fraction found only by the other crawlers, in the middle the lines found by both and on top are the results found by Black Widow. The bars are sorted by the difference of unique lines found by Black Widow and the other crawlers. Black Widow finds the highest number of unique lines of code in all cases except the rightmost, in which Enemy of the State performed better on Drupal. The exact number can be found in Table B.1.

Table B.3: Comparison of unique lines of code found by our scanner ($A \setminus B$) and the other scanners ($B \setminus A$). Improvement is new lines found by our scanner divided by the other's total.

Crawler	Our scanner $A \setminus B$	Other scanners $B \setminus A$	Other's total B	Improvement $\lvert A \setminus B \rvert / \lvert B \rvert$
Arachni	179 477	71 489	283 664	+63.3%
Enemy	267 372	25 268	149 548	+178.8%
jÄk	242 066	9 216	158 802	+152.4%
Skipfish	239 064	8 206	160 794	+148.7%
w3af	249 881	7 328	149 099	+167.6%
Wget	289 698	1 405	103 359	+280.3%
ZAP	226 088	5 560	171 124	+132.1%

Table B.4: Number of reported XSS injections by the scanners and the classification of the injection as either reflected or stored.

Crawler	Arachni		Enemy		jÄk		Skipfish		w3af		Widow		ZAP	
Type	R	S	R	S	R	S	R	S	R	S	R	S	R	S
Drupal	-	-	-	-	-	-	-	-	-	-	-	-	-	-
HotCRP	-	-	-	-	-	-	-	-	-	-	1	-	-	-
Joomla	-	-	8	-	-	-	-	-	-	-	-	-	-	-
osCommerce	-	-	-	-	-	-	-	-	-	-	1	1	9	-
phpBB	-	-	-	-	-	-	-	-	-	-	-	32	-	-
PrestaShop	-	-	-	-	-	-	-	-	-	-	2	-	-	-
SCARF	31	-	-	-	-	-	-	-	1	-	3	5	-	-
Vanilla	2	-	-	-	-	-	-	-	-	-	1	2	-	-
WackoPicko	3	1	2	1	13	-	1	1	1	-	3	2	-	-
WordPress	-	-	-	-	-	-	-	-	-	-	1	1	-	-

Table B.5: Number of unique and correct XSS injections by the scanners and the classification of the injection as either reflected or stored.

Crawler	Arachni		Enemy		jÄk		Skipfish		w3af		Widow		ZAP	
Type	R	S	R	S	R	S	R	S	R	S	R	S	R	S
Drupal	-	-	-	-	-	-	-	-	-	-	-	-	-	-
HotCRP	-	-	-	-	-	-	-	-	-	-	1	-	-	-
Joomla	-	-	-	-	-	-	-	-	-	-	-	-	-	-
osCommerce	-	-	-	-	-	-	-	-	-	-	1	1	-	-
phpBB	-	-	-	-	-	-	-	-	-	-	-	3	-	-
PrestaShop	-	-	-	-	-	-	-	-	-	-	1	-	-	-
SCARF	3	-	-	-	-	-	-	-	1	-	3	5	-	-
Vanilla	-	-	-	-	-	-	-	-	-	-	1	2	-	-
WackoPicko	3	1	2	1	1	-	1	-	1	-	3	2	-	-
WordPress	-	-	-	-	-	-	-	-	-	-	1	1	-	-

B.4.4 Code Injection Results

This section presents the results from the vulnerabilities the different scanners find. To be consistent with the terminology used in previous works [8, 30], we define an XSS vulnerability to be any injected JavaScript code that results in execution. While accepting JavaScript from users is risky in general, some applications, like Wordpress, have features which require executing user supplied JavaScript. In Section B.5.7 we discuss the impact and exploitability of the vulnerabilities our scanner finds.

In Table B.4 we list all the XSS vulnerabilities found by the scanners on all the applications. The table contains the number of self-reported vulnerabilities. After removing the false positives and clustering similar injections, as explained in Section B.4.2, we get the new results in Table B.5. The results from Table B.5 show that Black Widow outperforms the other scanners on both the reference applications and the modern applications. In total, Black Widow finds 25 unique vulnerabilities, which is more than 3 times as many as the second-best scanner. Of these 25, 6 are previously unknown vulnerabilities in modern applications. We consider the remaining 19 vulnerabilities to be known for the following reasons. First, all WackoPicko vulnerabilities are implanted by the authors and they are all known by design. Second, SCARF has been researched thoroughly and vulnerabilities may be already known. We conservatively assumed the eight vulnerabilities to be known. Third, the vulnerabilities on phpBB and Vanilla were fixed in their newest versions.

It is important to note we did not miss any vulnerability that the others found. However, there were cases where both Black Widow and other scanners found the same vulnerability but by injecting different parameters. We explore these cases more in Section B.5.6. Furthermore, Black Widow is the only scanner that finds vulnerabilities in the modern web applications.

B.4.5 Takeaways

We have shown that our scanner can outperform the other scanners in terms of both code coverage and vulnerability detection. Figure B.5 and Table B.1 show that we outperform the other scanners in 69 out of 70 cases. Additionally, Table B.2 and Table B.3 show we improve code coverage by between 63% and 280% compared to the other scanners and by between 6% and 62%, compared to the *sum* of all other scanners. We also improve vulnerability detection, as can be seen in Table B.4 and Table B.5. Not only do we match the other scanners but we also find new vulnerabilities in production applications.

In the next section, we will analyze these results closer and conclude which features allowed us to improve coverage and vulnerability detection. We also discuss what other scanners did better than us.

B.5 Analysis of Results

The results from the previous section show that the code coverage of our scanner outperforms the other ones. Furthermore, we find more code injections and in par-

ticular more stored XSS. In this section, we analyze the factors which led to our advantage. We also analyze where and why the other scanners performed worse.

Since we have access to the executed lines of code we can closely analyze the path of the scanner through the application. We utilize this to analyze when the scanners miss injectable parameters, what values they submit, when they fail to access parts of the application and how they handle sessions.

We start by presenting interesting cases from the code coverage evaluation in Section B.5.1, followed by an analysis of the reported vulnerabilities from all scanners in Section B.5.2. In Section B.5.3, we discuss the injections our scanner finds and compare it with what the others find. In Section B.5.4, we perform two case studies of vulnerabilities that only our scanner finds and which requires both workflow traversal and dependency analysis. Finally, in Section B.5.5, we extract the crucial features for finding injections based on all vulnerabilities that were found.

B.5.1 Coverage Analysis

As presented in Section B.4.3, Black Widow improved code coverage, compared to the aggregated result of all the other scanners, ranged from 5.5% on Drupal to 62% on PrestaShop. Comparing the code coverage to each scanner, Black Widow's improvement ranged from 63.3% against Arachni to 280% against Wget. In this section, we analyze the factors pertaining to code coverage by inspecting the performance of the different scanners. To better understand our performance we divide the analysis into two categories. We look at both cases where we have low coverage compared to the other scanners and cases where we have high relative coverage.

Low coverage As shown in Figure B.5, Enemy of the State is the only scanner that outperforms Black Widow and this is specifically on Drupal. Enemy of the State high coverage on Drupal is because it keeps the authenticated session state by avoiding logging out. The reason Black Widow lost the state too early was two-fold. First, we use a heuristic algorithm, as explained in Section B.3.2 to select the next edge and unfortunately the logout edge was picked early. Second, due to the structure of Drupal, our scanner did not manage to re-authenticate. In particular, this is because, in contrast to many other applications, Drupal does not present the user with a login form when they try to perform an unauthorized operation. To isolate the reason for the lower code coverage, we temporarily blacklist the Drupal logout function in our scanner. This resulted in our scanner producing similar coverage to Enemy of the State, ensuring the factor behind the discrepancy is session handling.

Skipfish performs very well on WordPress, which seems surprising since it is a modern application that makes heavy use of JavaScript. However, WordPress degrades gracefully without JavaScript, allowing scanners to find multiple pages without using JavaScript. Focusing on static pages can generate a large coverage but, as is evident from the detected vulnerabilities, does not imply high vulnerability detection.

High coverage Enemy of the State also performs worse against Black Widow on osCommerce and HotCRP. This is because Enemy of the State is seemingly entering an infinite loop, using 100% CPU without generating any requests. This could be due

to an implementation error or because the state inference becomes too complicated in these applications.

Although Black Widow performs well against Wget, Wget still finds some unique lines, which can seem surprising as it has previously been used as a reference tool [8, 30]. Based on the traces and source code, we see that most of the unique lines of code Wget finds are due to state differences, e.g. visiting the same page Black Widow finds but while being unauthenticated.

B.5.2 False positives and Clustering

To better understand the reason behind the false positives, and be transparent about our clustering, we analyze the vulnerabilities reported in Table B.4. For each scanner with false positives, we reflect on the reasons behind the incorrect classification and what improvements are required. We do not include w3af in the list as it did not produce any false positives or required any clustering.

a) Arachni reports two reflected XSS vulnerabilities in Vanilla. The injection point was a Cloudflare cookie used on the online support forum for the Vanilla web application. The cookie is never used in the application and we were unable to reproduce the injection. In addition, Arachni finds 31 XSS injections on SCARF. Many of these are incorrect because Arachni reuses payloads. For example, by injecting into the title of the page, all successive injection will be label as vulnerable.

b) Enemy of the State claims the discovery of 8 reflected XSS vulnerabilities on Joomla. However, after manual analysis, none of these result in code execution. The problem is that Enemy of the State interprets the reflected payload as an executed payload. It injects, `eval(print "[random]")`, into a search field and then detects that `"[random]"` is reflected. It incorrectly assumes this is because `eval` and `print` were executed. For this reason, we consider Enemy of the State to find 0 vulnerabilities on Joomla.

c) jÄk reports 13 vulnerabilities on WackoPicko. These 13 reports were different payloads used to attack the search parameter. After applying our clustering method, we consider jÄk to find one unique vulnerability.

d) Black Widow finds 32 stored vulnerabilities on phpBB. Most of these parameters are from the configuration panel and are all used in the same database query. Therefore, only 3 can be considered unique. Two parameters on PrestaShop are used in the same request, thus only one is considered unique. Black Widow did not produce any false positives thanks to our dynamic detection method explained in Section B.3.4

e) Skipfish claims the detection of a stored XSS in WackoPicko in the image data parameter when uploading an image. However, the injected JavaScript could not be executed. Interesting to note is that Skipfish was able to inject JavaScript into the guestbook but was not able to detect it.

f) ZAP claims to find 9 reflected XSS injection on osCommerce. They are all variations of injecting `javascript:alert(1)` into the parameter of a link. Since it was just part of a parameter and not a full URL, the JavaScript code will never execute. Thus, all 9 injections were false positives.

B.5.3 What We Find

In this section, we present the XSS injections our scanner finds in the different applications. We also extract the important features which made it possible to find them.

HotCRP: Reflected XSS in bulk user upload The admin can upload a file with users to add them in bulk. The name of the file is then reflected on the upload page. To find this, the scanner must be able to follow a complex workflow that makes heavy use of JavaScript, as well as handle file parameters. It is worth noting that the filename is escaped on other pages in HotCRP but missed in this case.

osCommerce; Stored and reflected XSS Admins can change the tax classes in osCommerce and two parameters are not correctly filtered, resulting in stored XSS vulnerabilities. The main challenge to find this vulnerability was to find the injection point as this required us to interact with a navigation bar that made heavy use of JavaScript.

We also found three vulnerable parameters on the review page. These parameters were part of a form and their types were *radio* and *hidden*. This highlights that we still inject all parameters, even if they are not intended to be changed.

phpBB; Multiple Stored XSS in admin backend Admins can change multiple different application settings on the configuration page, such as flooding interval for posts and max avatar file size. On a separate page, they can also change the rank of the admin to a custom title. In total, this results in 32 vulnerable parameters that can be clustered to 3 unique ones. These require inter-state dependency analysis to solve. Once a setting is changed, the admin is met with a "Successful update" message, which does not reflect the injection. Thus, the dependency must be found to allow for successful fuzzing.

PrestaShop; Reflected XSS in admin dashboard The admin dashboard allows the admin to specify a date range for showing statistics. Two parameters in this form are not correctly filtered and result in a reflected XSS. Finding these requires a combination of modeling JavaScript events and handling workflows. To find this form the scanner must first click on a button on the dashboard.

SCARF; Stored XSS in comments There are many vulnerabilities in SCARF, most are quite easy to find. Instead of mentioning all, we focus on one that requires complex workflows, inter-state dependencies and was only found by us. The message field in the comment section of conference papers is vulnerable. What makes it hard to find is the traversing and needed before posting the comment and the inter-state dependency analysis needed to find the reflection. The scanner must first create a user, then create a conference, after which it can upload a paper that can be commented on.

Vanilla; Stored and reflected XSS The language tag for the RSS feed is vulnerable and only reflected in the feed. Note that the feed is served as HTML, allowing JavaScript to execute. There is also a stored vulnerability in the comment section which can be executed by saving a comment as a draft and then viewing it. Both of

these require inter-state dependency analysis to find the connecting between language settings and RSS feeds, as well as posting comments and viewing drafts.

Black Widow also found a reflected XSS title parameter in the configuration panel that was vulnerable. Finding this mainly required and modeling JavaScript and forms.

WackoPicko; Multi-step stored XSS We found all the known XSS vulnerabilities [7], except the one requiring flash as we consider it out-of-scope. We also found a non-listed XSS vulnerability in the reflection of a SQL error. Most notably we were able to detect the multi-step XSS vulnerability that no other scanner could. This was thanks to both inter-state dependency tracking and handling workflows. We discuss this in more detail in the case study in Section B.5.4.1.

WordPress; Stored and reflected XSS The admin can search for nearby events using the admin dashboard. The problem is that the search query is reflected, through AJAX, for the text-to-speech functionality. Finding this requires modeling of both JavaScript events, network requests and forms.

Our scanner also found that by posting comments from the admin panel JavaScript is allowed to run on posts. For this, the scanner must handle the workflows needed to post the comments and the inter-state dependency analysis needed to later find the comment on a post.

B.5.4 Case Studies

In this section, we present two in-depth case studies of vulnerabilities that highlights how and why our approach finds vulnerabilities the other scanners do not. We base our analysis on server-side traces, containing the executed lines of code, generated from the scanner sessions. By manually analyzing the source code of an application we can determine the exact lines of code that need to be executed for an injection to be successful.

The cases we use are the comment section in WackoPicko and the configuration panel in phpBB. As we have shown, Black Widow can find vulnerabilities in more complex modern web applications. Nevertheless, these cases allow us to limit the number of factors when comparing our approach with the other scanners. Since WackoPicko and phpBB have been used in previous studies [8, 30] they also serve as a level playing field for all scanners.

B.5.4.1 Comments on WackoPicko

WackoPicko has a previously unsolved multistep XSS vulnerability that no other scanner has been able to find. The difficultly of finding and exploiting is the need for correctly reproducing a specific workflow. After submitting a comment via a form the user needs to review the comment. While reviewing, the user can choose to either delete the comment or add it. If, however, the user decided to visit another page, before adding or deleting, then the review form will be removed and the user will have to resubmit the comment before reviewing it again. Thus, the steps that must be taken are: Find an image to comment on (`view.php#50`, i.e. line 50

Table B.6: Steps to recreate the vulnerability in WackoPicko. The columns contain the file name and line of code for each step.

Crawler	view.php#50	preview_comment.php#54	view.php#53	Exploit
Arachni	✓	✓	✓	
Enemy	✓	✓	✓	
jÄk				
Skipfish	✓			
w3af	✓			
Widow	✓	✓	✓	✓
ZAP				

in `view.php`), Post a comment (`preview_comment.php#54`), Accept the comment while reviewing (`view.php#53`) In Table B.6 we note that two scanners are able to find the input but not exploit it.

Both Enemy of the State and Arachni managed to post a comment but neither could exploit the vulnerability. Enemy of the State was able to post a comment containing an empty string but the fuzzing was unsuccessful. Arguably, Arachni made it a bit further since it was able to inject an XSS payload. However, the payload was not detected and reported. Enemy of the State's shortcoming is that it fuzzes the forms independently while Arachni's shortcoming is that it forgets it's own injection.

jÄk and ZAP had problems finding the first step, i.e. viewing the pictures, because the login form breaks the HTML standard by putting a form inside a table [41]. We avoid this by using a modern browser to parse the web page. This allows Black Widow to view the web page as the developer intended, assuming they tested it in a modern browser

Both w3af and Skipfish were able to find the pictures but not able to post the comment. w3af because it could not model the `textarea` in the form. Skipfish, on the other hand, does not have this problem. We believe that Skipfish logged out after seeing the picture but before posting the comment. The data shows that Skipfish does not try to log in multiple times. In comparison, we correctly handle the `textarea` allowing us to post comments. At the same time, we also try to log in multiple times if presented with a login form. This mitigates losing the session forever at an early stage.

To solve this challenge Black Widow needs to combine the modeling of form elements, handle workflows and use inter-state dependency analysis to correctly inject and detect the vulnerability.

B.5.4.2 Configuration on phpBB

The configuration panel on phpBB has multiple code injection possibilities. To find these the crawler must overcome two challenges. First, to reach the admin panel requires two logins, the first to authenticate as a user and then again, with the same credentials, to authenticate as an administrator. Second, the injected param-

Table B.7: Steps to recreate the vulnerability in phpBB. The columns contain the file name and line of code for each step.

Crawler	admin/ index.php#28	admin_ board.php#34	admin_ board.php#74	admin_ board.php#34	Exploit
Arachni					
Enemy					
jÄk					
Skipfish					
w3af					
Widow	✓	✓	✓	✓	✓
ZAP					

eter is not reflected on the same page. To detect this injection inter-state dependency analysis is required. The steps needed to find the vulnerability is, log in as admin (`admin/index.php#28`), find the vulnerable form (`admin_board.php#34`), successfully update the database (`admin_board.php#74`) find the reflection of the input (`admin_board.php#34`).

As shown in Table B.7, none of the other scanners managed to access the configuration panel. This is because phpBB requires a double login. Arachni, jÄk, Skipfish, w3afand ZAP all require user-supplied credentials together with parameters before running. Based on the traces they do not try these credentials on the admin login form, only the first login form. Enemy of the State, on the other hand, tries the standard username and password `scanner1`. This was enough to log in but it did not manage to log in as an admin.

Our scanner solves the double login by being consistent with the values we submit. This allows us to both authenticate as a user and then also as an admin when presented with the login prompt. After submitting the form in configuration panel with our taint tokens and later revisiting it, we detect the inter-state dependency and can fuzz the source and sink.

B.5.5 Features Attribution

In this section, we identify and attribute the key features that contributed to finding the vulnerabilities in the web applications.

In particular, we try to determine the impact of our modeling, traversing and inter-state dependency analysis techniques. Below are the definitions we use in Table B.8.

Modeling Modeling is considered to contribute if a combination of HTML forms and JavaScript events were used to find the code injection.

Traversal Workflow traversal contributes if the point of injection depends on a previous state. This could, for example, be a form submission, a click of a button or some other DOM interaction.

Table B.8: For each of the vulnerabilities we note contributing features, i.e. modeling, workflow reproduction or inter-state dependency (ISD) analysis. We also present if they were uniquely detected by Black Widow.

Id	Application	Description	Model	Workflow	ISD	Unique
1	HotCRP	User upload	✓	✓		✓
2	osCommerce	Review rating				✓
3	osCommerce	Tax class				✓
4	phpBB	Admin ranks			✓	✓
5	phpBB	Configuration			✓	✓
6	phpBB	Site name			✓	✓
7	PrestaShop	Date	✓	✓		✓
8	SCARF	Add session		✓	✓	✓
9	SCARF	Comment		✓	✓	✓
10	SCARF	Conference name				
11	SCARF	Edit paper		✓	✓	✓
12	SCARF	Edit session				
13	SCARF	Delete comment		✓	✓	✓
14	SCARF	General options				
15	SCARF	User options			✓	
16	Vanilla	Comment draft			✓	✓
17	Vanilla	Locale			✓	✓
18	Vanilla	Title banner	✓			✓
19	WackoPicko	Comment				
20	WackoPicko	Multi-step		✓	✓	✓
21	WackoPicko	Picture				
22	WackoPicko	Search				
23	WackoPicko	SQL error				
24	WordPress	Comment		✓	✓	✓
25	WordPress	Nearby event	✓	✓	✓	✓

Inter-state dependency A code injection is defined to need inter-state dependency analysis if the point of reflection is different from the point of injection.

Table B.8 shows the 25 unique code injections from the evaluation. Of these, modeling contributed to 4, workflow traversal contributed to 9, and inter-state dependency analysis contributed to 13. In total, at least one of them was a contributor in 16 unique injections. The remaining 9 were usually simpler. Four of them were from WackoPicko where the results of injection were directly reflected. SCARF had 3 directly reflected injections and osCommerce had 2. It is clear, especially for unique vulnerabilities, that modeling, workflow traversal and inter-state dependency analysis plays an important role in detecting stored XSS vulnerabilities.

B.5.6 Missed by Us

Out of the 25 unique injections found by all scanners, we also find all 25. There was, however, an instance where Arachni found a vulnerability by injecting a different parameter than we did. This does not constitute a unique vulnerability due to our clustering, which we explain in Section B.4.4. On SCARF, input elements can

be dynamically generated by adding more users. The input names will simply be `1_name`, `2_name`, etc. Arachni managed to add multiple users by randomizing email addresses. Since our crawler is focused on consistency, we do not generate valid random email addresses and could therefore not add more than one user.

The drawback, as we have discussed is that is it easier to lose the state if too much randomness is used. A possible solution to this could be to keep two sets of default values and always test both when possible. There is still the risk that using multiple users can result in mixing up the state between them. It would also introduce a performance penalty as multiple submissions for each form would be required.

The w3af scanner was able to find a reflected version of a vulnerable parameter that we considered to be stored. In this particular case on SCARF, it was possible to get a direct reflection by submitting the same *password* and *retype password* in the user settings. This is what w3af did. Our scanner injected unique values into each field, resulting in an error without reflection, however, the fields were still stored. Inter-state dependency analysis was used to detect these stored values when revisiting the user settings.

Further possible improvements include updating our method for determining safe requests and more robust function hooking. A machine learning approach, such as Mitch [6], could be used to determine if a request can be considered safe. The function hooking could be done by modifying the JavaScript engine instead of instrumenting JavaScript code.

B.5.7 Vulnerability Exploitability

For the six new vulnerabilities, we further investigate the impact and exploitability. While all of these vulnerabilities were found using an admin account in the web application, the attacker does not necessarily need to be an admin. In fact, XSS payloads executed as the admin gives a higher impact as the JavaScript runs with admin privileges. What the attacker needs to do is usually to convince the admin to click on a link or visit a malicious website, i.e. the attacker does not require any admin privileges. Although, there might be an XSS vulnerability in the code, i.e. user input being reflected, there are orthogonal mitigations such as CSRF tokens and CSP that can decrease the exploitability.

To exploit the HotCRP vulnerability the attacker would have to guess a CSRF token, which is considered difficult. Similarly, PrestaShop has a persistent secret in the URL which would have to be known by the attacker. One of the WordPress vulnerabilities was a self-XSS, meaning the admin would need to be convinced to, in this case, input our payload string, while the other one required a CSRF token. Finally, osCommerce required no CSRF tokens making it both high impact and easy to exploit.

B.5.8 Coordinated Disclosure

We have reported the vulnerabilities to the affected vendors, following the best practices of coordinated disclosure [15]. Specifically, we reported a total of six vulnerabilities to HotCRP, osCommerce, PrestaShop and WordPress.

So far our reports have resulted in HotCRP patching their vulnerability [24]. A parallel disclosure for the same vulnerability was reported to PrestaShop and is now tracked as CVE-2020-5271 [1]. Due to the difficulty of exploitation, WordPress did not consider them vulnerabilities. However, the *nearby event* vulnerability is fixed in the latest version. We have not received any confirmation from osCommerce yet.

B.6 Related Work

This section discusses related work. Automatic vulnerability scanning has been a popular topic due to its complexity and practical usefulness. This paper focuses on blackbox scanning, which requires no access to the application's source code or any other input from developers. We have evaluated our approach with respect to both community-developed open-source tools [28, 34, 36] and academic blackbox scanners [8, 30]. There are also earlier works on vulnerability detection and scanning [2, 10, 11, 17, 23, 35]. While we focus on blackbox testing, there is also progress on whitebox security testing [13, 18, 22, 25, 39].

As previous evaluations [3, 9, 29, 37, 40] show, detecting stored XSS is hard. A common notion is that it is not the exact payload that is the problem for scanners but rather crawling deep enough to find the injections, as well as, model the application to find the reflections. Similar to our findings, Parvez et al. [29] note that while some scanners were able to post comments to pictures in WackoPicko, something which requires multiple actions in sequence, none of them was able to inject a payload.

We now discuss work that addresses server-side state, client-side state, and tracking data dependencies.

Server-side state Enemy of the State [8] focuses on inferring the state of the server by using a heuristic method to compare how requests result in different links on pages. Black Widow instead takes the approach of analyzing the navigation methods to infer some state information. For example, if the previous edge in the navigation graph was a form submission then we would have to resubmit this form before continuing. This allows us to execute sequences of actions without fully inferring the server-side state.

One reason many of the other scanners pay little attention to server-side state is to prioritize performance from concurrent requests. Skipfish [42] is noteworthy for its high performance in terms of requests per second. One method they use to achieve this is making concurrent requests. Concurrent requests can be useful in a stateless environment since the requests will not interfere with each other. ZAP [28], w3af [34] and Arachni [36] take the same approach as Skipfish and use concurrent requests in favor of better state control. Since our traversing method relies on executing a sequence of possibly state-changing action we need to ensure that no other

state-changing requests are sent concurrently. For this reason, our approach only performs actions in serial.

Client-side state jÄk considers client-side events to improve exploration. The support for events is however limited, leaving out such events as form submission. While other scanners like Enemy of the State, w3af, and ZAP execute JavaScript, they do not model the events. This limits their ability to explore the client-side state. As modern applications make heavy use of JavaScript, Black Widow offers fully-fledged support of client-side events. In contrast to jÄk, Black Widow models client-side events like any other navigation method. This means that we do not have to execute the events in any particular order which allows us to chain them with other navigations such as form submissions.

Tracking data dependencies Tracking payloads is an important part of detecting stored XSS vulnerabilities. Some scanners, including Arachni, use a session-based ID in each payload. Since the ID is based on the session this can lead to false positives as payloads are reused for different parameters. jÄk and Enemy of the State use unique IDs for their payload but forgets them on new pages. w3af uses unique payloads and remembers them across pages. ZAP uses a combination in which a unique ID is sent together with a generic payload but in separate requests. This works if both the ID and payload are stored on a page. In addition to using unique IDs for all our payloads, Black Widow incorporates the inter-state dependencies in the application to ensure that we can fuzz the correct input and output *across* different pages.

LigRE [10], and its successor KameleonFuzz [11] use a blackbox approach to reverse engineering the application and apply a genetic algorithm to modify the payloads. While they also use tainting inside the payloads to track them, we use plaintext tokens to avoid filters destroying the taints. While Black Widow works on live applications, KameleonFuzz requires the ability to reset the application. Unfortunately, neither LigRE nor KameleonFuzz are open-source, which has hindered us from their experimental evaluation.

B.7 Conclusion

We have put a spotlight on key challenges for crawling and scanning the modern web. Based on these challenges, we have identified three core pillars for deep crawling and scanning: navigation modeling, traversing, and tracking inter-state dependencies. We have presented Black Widow, a novel approach to blackbox web application scanning that leverages these pillars by developing and combining augmented navigation graphs, workflow traversal, and inter-state data dependency analysis. To evaluate our approach, we have implemented it and tested it on 10 different web applications and against 7 other web application scanners. Our approach results in code coverage improvements ranging from 63% to 280% compared to other scanners across all tested applications. Across all tested web applications, our approach improved code coverage by between 6% and 62%, compared to the *sum* of all other scanners. When deployed to scan for cross-site scripting vulnerabilities, our approach has featured no false positives while uncovering more vulnerabilities than the other

scanners, both in the reference applications, i.e. phpBB, SCARF, Vanilla and WackoPicko, and in production software, including HotCRP, osCommerce, PrestaShop and WordPress.

Acknowledgment

We would like to thank Sebastian Lekies for inspiring discussions on the challenges of web scanning. We would also like to thank Nick Nikiforakis and the reviewers for their valuable feedback. This work was partially supported by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

Bibliography

[1] CVE-2020-5271. Available from MITRE, CVE-ID CVE-2020-5271., Apr. 20 2020.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401. IEEE, 2008.

[3] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy*, pages 332–345. IEEE, 2010.

[4] C.-P. Bezemer, A. Mesbah, and A. van Deursen. Automated security testing of web widget interactions. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 81–90. ACM, 2009.

[5] Bugcrowd. The State of Crowdsourced Security in 2019. `https://www.bugcrowd.com/`, 2020.

[6] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei. Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 528–543. IEEE, 2019.

[7] A. Doupé. Wackopicko, 2018.

[8] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *USENIX Security Symposium 12*, pages 523–538, 2012.

[9] A. Doupé, M. Cova, and G. Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131. Springer, 2010.

[10] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz. Ligre: Reverse-engineering of control and data flow models for black-box xss detection. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 252–261. IEEE, 2013.

[11] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 37–48, 2014.

[12] Facebook. A Look Back at 2019 Bug Bounty Highlights. `https://www.facebook.com/notes/facebook-bug-bounty/a-look-back-at-2019-bug-bounty-highlights/3231769013503969/`, 2020.

[13] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, volume 58, 2010.

[14] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, RFC Editor, June 2014.

[15] Google. Project zero: Vulnerability disclosure faq, 2019.

[16] Google. Vulnerability Reward Program: 2019 Year in Review. `https://security.googleblog.com/2020/01/vulnerability-reward-program-2019-year.html`, 2020.

[17] W. G. Halfond, S. R. Choudhary, and A. Orso. Penetration testing with improved input vector identification. In *2009 International Conference on Software Testing Verification and Validation*, pages 346–355. IEEE, 2009.

[18] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52, 2004.

[19] S. Idrissi, N. Berbiche, F. Guerouate, and M. Shibi. Performance evaluation of web application security scanners for prevention and protection against vulnerabilities. *International Journal of Applied Engineering Research*, 12(21):11068–11076, 2017.

[20] InfoSecurity. XSS is Most Rewarding Bug Bounty as CSRF is Revived. `https://www.infosecurity-magazine.com/news/xss-bug-bounty-csrf-1-1-1-1/`, 2019.

[21] S. Innovation. Google Awards $1.2 Million in Bounties Just for XSS Bugs. `https://blog.securityinnovation.com/google-awards-1.2-million-in-bounties-just-for-xss-bugs`, 2016.

[22] N. Jovanovic, C. Kruegel, and E. Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010.

[23] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, pages 247–256, 2006.

[24] E. Kohler. Correct missing quoting reported by Benjamin Eriksson at Chalmers. `https://github.com/kohler/hotcrp/commit/81b7ffee2c5bd465c82acf139cc064daacca845c`, 2020.

[25] X. Li, W. Yan, and Y. Xue. Sentinel: securing database from logic flaws in web applications. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 25–36, 2012.

[26] A. Mesbah, E. Bozdag, and A. Van Deursen. Crawling ajax by inferring user interface state changes. In *2008 Eighth International Conference on Web Engineering*, pages 122–134. IEEE, 2008.

[27] H. Nikšić. Wget - gnnu project, 2019.

[28] OWASP. Owasp zed attack proxy (zap), 2020.

[29] M. Parvez, P. Zavarsky, and N. Khoury. Analysis of effectiveness of black-box web application scanners in detection of stored sql injection and stored xss vulnerabilities. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 186–191. IEEE, 2015.

[30] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *International Symposium on Recent Advances in Intrusion Detection*, pages 295–316. Springer, 2015.

[31] A. Petukhov and D. Kozlov. Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing. *Computing Systems Lab, Department of Computer Science, Moscow State University*, pages 1–120, 2008.

[32] PortSwigger. Burp Scanner - PortSwigger. `https://portswigger.net/burp/documentation/scanner`, 2020.

[33] D. Rethans. Xdebug - debugger ad profiler tool for php, 2019.

[34] A. Riancho. w3af - open source web application security scanner, 2007.

[35] T. S. Rocha and E. Souto. Etssdetector: A tool to automatically detect cross-site scripting vulnerabilities. In *2014 IEEE 13th International Symposium on Network Computing and Applications*, pages 306–309, Aug 2014.

[36] Sarosys LLC. Framework - arachni - web application security scanner framework, 2019.

[37] L. Suto. Analyzing the accuracy and time costs of web application security scanners. *San Francisco, February*, 2010.

[38] The OWASP Foundation. Owasp top 10 - 2017, 2017. `https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf`.

[39] A. Vernotte, F. Dadeau, F. Lebeau, B. Legeard, F. Peureux, and F. Piat. Efficient detection of multi-step cross-site scripting vulnerabilities. In A. Prakash and R. Shyamasundar, editors, *Information Systems Security*, pages 358–377, Cham, 2014. Springer International Publishing.

[40] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 566–571. IEEE, 2009.

[41] WHATWG. Html standard, 2019.

[42] M. Zalewski. Skipfish, 2015.

Appendix

B.I Scanner configuration

B.I.1 Arachni

The following command was used to run Arachni.

```
arachni [url] --check=xss* --browser-cluster-pool-size=1 --plugin?autologin:url
    =[loginUrl],parameters="[userField]=[username]&[passField]=[password]",
    check="[logout string]}
```

B.I.2 Black Widow

The following command was used to run Black Widow.

```
python3 crawl.py [url]
```

B.I.3 Enemy of the State

First we changed the username and password in the web application to *scanner1* then we ran the following command.

```
jython crawler2.py [url]
```

B.I.4 jÄk

We updated the `example.py` file with the URL and user data.

```
url = [url]
user = User("[sessionName]", 0, url, login_data = {"[userField]": "[username]",
     "[passField]": "[password]"}, session="ABC")
```

B.I.5 Skipfish

The following command was used to run Skipfish.

```
skipfish -uv -o [output]
         --auth-form [loginUrl]
         --auth-user-field [userField]
         --auth-pass-field [passField]
         --auth-user [username]
         --auth-pass [password]
         --auth-verify-url [verifyUrl]
         [url]
```

B.I.6 w3af

For w3af we used the following settings, *generic* and *xss* for the audit plugin, *web_spider* for crawl plugin and *generic* (with all credentials) for the auth plugin.

B.I.7 Wget

The following command was used to run Wget.

```
1 wget -rp -w 0 waitretry=0 -nd --delete-after --execute robots=off [url]
```

B.I.8 ZAP

For ZAP we used the *automated scan* with both *traditional spider* and *ajax spider*. In the *Scan Progress* window we deactivated everything that was not XSS. Similar to Enemy of the State, we changed the credentials in the web application to the scanner's default, i.e. *ZAP*.

C

Black Ostrich: Web Application Scanning with String Solvers

Abstract. Securing web applications remains a pressing challenge. Unfortunately, the state of the art in web crawling and security scanning still falls short of deep crawling. A major roadblock is the crawlers' limited ability to pass input validation checks when web applications require data of a certain format, such as email, phone number, or zip code. This paper develops Black Ostrich, a principled approach to deep web crawling and scanning. The key idea is to augment web crawling, based on the Black Widow tool, with string constraint solving to dynamically infer suitable inputs from regular expression patterns in web applications, and thereby pass input validation checks. To enable this use of constraint solvers, we develop new automata-based techniques to handle complex real-world regular expressions, including support for the relevant features of ECMA JavaScript regular expressions, and implement those methods in the Ostrich solver. We evaluate Black Ostrich on a set of 8 821 unique validation patterns gathered from over 21 667 978 forms from a combination of the July 2021 Common Crawl and Tranco top 100K. For these forms and reconstructions of input elements corresponding to the `patterns`, we demonstrate that Black Ostrich achieves a 99% coverage of the form validations compared to an average of 36% for the state-of-the-art scanners, while also yielding a 45% increase for vulnerability detection. We further show that our approach can boost coverage by evaluating it on three open-source applications. Our empirical studies include a study of email validation patterns, simultaneously demonstrating that our regular expression encoding is practical, where we find that many (213/825) of the patterns are susceptible to trivial XSS injection attacks.

C.1 Introduction

As the modern digitalized society increasingly relies on web applications, securing them remains an important challenge Web security scanners like Arachniand ZAPplay an important role, focusing on crawling and scanning for vulnerabilities. Recent efforts by the research community have focused on moving away from traditional static crawling techniques based on link discovery and URL traversal. As JavaScript enables increasingly dynamic and complex web pages, new approaches

incorporate dynamic behaviors as in jÄk [39] and asynchronous HTTP requests as in CrawlJAX [6, 37]. Other approaches address the complexity of the server-side application by reverse engineering, as in LigRE [16] and KameleonFuzz [17], or inferring the state of the server as in Enemy of the State [15] to use the learned model for driving a crawler. Black Widow [18] demonstrates how to fruitfully combine navigation modeling, traversing, and tracking inter-state dependencies for black-box web application scanning.

Web scanning challenges. While this progress is encouraging, unfortunately, the state of the art in web crawling and security scanning still falls short of deep crawling. A major roadblock is the crawlers' limited ability to pass input validation checks when web applications require data of a certain format, such as email, phone number, or zip code.

Understanding what type of data to send is crucial for exploring more of the application but also for generating valid payloads. Consider the pattern `.*@.*\.[a-z]{2,3}` for emails, which allows *anything* followed by an `@`-sign followed by *anything* followed by a period followed by two or three lowercase letters. This expression will match a valid email address, but also invalid, potentiality malicious, ones, like `<script>alert(1)</script>@mail.com`. For a scanner to find this vulnerability it needs to be able to generate a payload that also matches the pattern. To do so using brute force is slow or practically impossible, and to use a library of prepared payloads (as many scanners do) is intractable since some sites use specialized validation. For example, we did not find any scanner with payloads matching the real-world pattern `.*France`.

Black Ostrich to the rescue. This paper proposes Black Ostrich, a principled approach to deep web crawling and scanning. The key idea is to leverage *string-based constraint solving, based on satisfiability modulo theories (SMT)* [14], to infer suitable input from the analysis of forms in web applications, including both input types, such as email and URL, and support for regular expression (or *regex*) patterns, thereby automatically passing input validation checks. While SMT is heavier than simply picking inputs from a library, we can instead be efficient in the number of network requests needed. SMT has been extensively used for web security for applications like finding SQL injections [31], analyzing and testing JavaScript [42], and detecting server-side parameter tampering [8]. However, these approaches largely focus on detecting vulnerabilities rather than the depth of web crawling. To the best of our knowledge, Black Ostrich is the *first to leverage SMT technology for deep web crawling*. As such, it requires addressing several research challenges.

SMT challenges. Several SMT solvers have been recently extended to *string constraints,* motivated by applications in security such as symbolic execution of string processing programs. This includes different versions of Z3 [13], Z3-str/2/3/4 [46], S3/p/# [44], CVC4 [35], Norn [1], Sloth [26], and Ostrich [10]. Yet the constraints in web scanning differ from the ones in typical verification. The main challenge is to handle *real-world regular expression constraints,* which can be extremely complex in web applications. Regular expressions used for input validation can be thousands of characters long, and frequently use extensions like *anchors* or *look-arounds*. From

our experiments we find a 500 using look-arounds and 4044 using anchors. The longest pattern we find is a stunning 29 059 characters.

Although SMT solvers for strings can handle textbook regular expressions [27], up to now no SMT solver directly supports the much richer language of real-world regular expressions [25].

Encoding the semantics of a modern regular expression engine into constraints for an SMT solver is highly non-trivial. To our knowledge, the only such translation was presented by Loring et al. [36] in the scope of symbolic execution by the ExpoSE tool. The requirements in web crawling are largely orthogonal, however, to the ones in symbolic execution: while Loring et al. [36] focus on correct handling of matching, back-references, and greediness, their support of the very commonly used feature of look-arounds is only partial (we provide a detailed comparison in Section C.3.4), and our experiments show that the SMT-LIB encoding in ExpoSE turns out to be a less natural match for the intricate regexes on the web (Section C.6.3).

Validation-aware crawling and fuzzing. When a crawler is faced with a form it must decide on what data it should submit. The type of data expected by the server might be numeric or a string. In addition to data type, other constraints can be put on the data, for example, the value needs to be a valid email or URL. Validation of such constraints can happen both client-side, within the browser, or server-side. To make progress in crawling, it is necessary to pass server-side checks, since the provided input will otherwise be rejected by the web application. Traditional scanners, therefore, source the required input data from a library with a diverse set of strings, hoping that one will pass the validation constraints. This is a problem as the approach is noisy, inefficient, and in the worst-case ineffective, since web applications can have arbitrarily complicated input validation.

Complementary to the traditional techniques, Black Ostrich applies a dynamic approach that takes all available information on the expected input data into account and systematically constructs input data through constraint solving. The first difficulty is the fact that server-side validation constraints are not visible, and can only be guessed by the crawler. Fortunately, modern HTML features come to our rescue: HTML5 provides several attributes for client-side input validation (Table C.5 in Section C.VII), including a new attribute called *patterns*. Patterns are regular expressions, as specified by the ECMA JavaScript standard [25]), that user input must match before the form can be submitted, and are today widely used in web applications.

Black Ostrich thus focuses on passing client-side validation constraints, with the assumption that successful inputs are likely to also satisfy server-side constraints. The presence of a vulnerable (e.g. XSS-accepting) pattern on a web page is by itself not a vulnerability, as web applications might enforce stricter server-side checks. Yet there is no natural reason for a web page to use different constraints for client-side and server-side checks, which is why Black Ostrich also uses string solving to produce payloads matching the patterns.

For JavaScript, we can dynamically extract regex tests on our inputs and update the inputs accordingly before submission. This works well for custom regex-based JavaScript validations. Many popular validation libraries, including jQuery Validate,

also rely on regex to validate predefined types such as email, meaning we can extract and solve it. However, JavaScript can use other methods for validation outside our coverage. That being said, due to the ease of use of HMTL5 patterns, we expect them to become increasingly common in the future, as they are indeed designed to replace JavaScript validation.

SMT solving. Black Ostrich dynamically generates input data for web pages both for exploring and fuzzing web pages. For this purpose, we define a translation of the HTML5 validation constraints (Table C.5 in Section C.VII) to logical SMT formulas, which can then be processed by an SMT solver to construct data that will pass all constraints, or show that no such data exists. Depending on the phase of scanning, the validation constraints can be complemented by constraints that ensure uniqueness of the input data, to discover dependencies in web applications, or by constraints that request the inclusion of *payloads* like `<script>alert(1)</script>` in the input.

Our starting point is the SMT solver Ostrich [10], an automata-based string solver for deciding the satisfiability of constraints in a rich language, including regular expressions, equations, and string functions including replace-all and letter-to-letter transduction. Prior to our work, and in line with other SMT solvers, Ostrich could only process regexes in SMT-LIB notation, and did not support ECMAScript [25] features such as anchors or look-arounds. This paper extends Ostrich with a native parser for ECMAScript regexes and proposes two new methods to handle ECMAScript regexes (including all features but back-references, see Section C.3.3) in an SMT solver: a lightweight, but partial translation from ECMAScript regexes to standard regexes, and a complete approach that works by translating ECMAScript regexes to two-way alternating automata. Completing the pipeline, we also present a new technique to simulate two-way alternating finite automata by non-deterministic finite automata that enables efficient implementation inside solvers.

Evaluation. We demonstrate that Black Ostrich boosts both code coverage and vulnerability detection, compared to state-of-the-art crawlers/scanners including Arachni, Enemy of the State, jÄk, ZAP, and Black Widow. For ethical reasons, we create a testbed, based on real-world forms, for the evaluation instead of directly running the scanners on real websites. Even for just coverage, running a scanner without the attack module can cause damage to the website in the form of forum posts, product reviews, purchases, etc.

To test Black Ostrich in a realistic environment without interfering with real websites, we harvest both forms and, separately, input validation patterns from the July 2021 Common Crawl archive consisting of 3.15 billion web pages [11]. Due to resource constraints, for forms, we sample uniformly from the 64 000 archive parts of the same archive, collecting forms from 8 266 577 URLs, in total 21 667 978 forms. To also capture validations used on popular websites, we combine this with a crawl of Tranco [34] top 100K.

Using the combined data from Common Crawl and Tranco we extract 881 329 HTML5 patterns which after de-duplication results in 9 805 patterns, all being used in the wild. After removing broken and invalid patterns we have a total of 8 821.

We create a testbed of mock websites using these patterns both on the client-side and server-side and evaluate the coverage for the state-of-the-art web crawlers. Our scanner shows a significant improvement by being able to solve 99% of the patterns compared to an average of 36% for the other scanners.

For vulnerability detection, we use the same testbed and include an input reflection if the server-side check is passed, resulting in an XSS vulnerability. The results show an increase of 45% for vulnerability detection compared to the other scanners. We find 828 vulnerable patterns compared to an average of 336 vulnerable patterns for the other scanners.

Open-source software. We explore the use of patterns in open-source web applications from GitHub. We download over 900 projects under the web applications topic and analyze their use of patterns. We perform a case study analysis on three applications that uses both client-side and server-side validation. In this study, we do a head-to-head comparison of all the scanners and show that we can increase coverage by passing input validation.

Email pattern study. We report on an empirical study of 825 email patterns extracted from the Common Crawl dataset of real-world web pages. Following the intuition that validation patterns are often shared between front-end and back-ends we discover that reusing client-side patterns on the server-side would then cause at least 502 of the 825 collected patterns to accept a trivial XSS attack with an embedded `<script>` tag. Furthermore, we show that at least 213 of the patterns would accept the same attack string even under the stronger semantics of the HTML5 `pattern` attribute. These experiments illustrate how our regular expression semantics encoding is versatile and efficient enough to handle complex real-world regular expressions for practical applications.

The contributions of the paper are:

- We develop a novel platform for validation-aware web crawling and scanning (Section C.2).
- We propose a new version of two-way alternating finite-state automata, 2AFA_{SMT} (Section C.3), and a simple yet efficient simulation of 2AFA_{SMT} using standard non-deterministic finite-state automata (Section C.4).
- Based on 2AFA_{SMT}, we define a new translation from regular expressions to NFA that provides complete coverage of the ECMAScript regex features, with exception of back-references. This translation enables us to extend the open-source solver Ostrich with native support for ECMA regular expressions (Section C.3).
- We evaluate the coverage and vulnerability detection (Section D.4), showing that our scanner solves 99% of the patterns compared to the average of 36% for the other scanners, a 175% improvement. We improve the detection of vulnerable patterns by 45% (Section E.4).
- We investigate the usage of HTML patterns in open-source web applications and demonstrate increased coverage thanks to string solving. (Section C.7).
- We present a case study of email validation patterns, pointing out common vulnerabilities and inconsistencies related to email patterns on the web (Section C.II).

C.2 Validation-aware Scanning

To improve the coverage and vulnerability detection rate of scanners we propose a design in which the scanner consults a string solver before submitting inputs. The string solver is used to generate strings matching the pattern, allowing the scanner to submit the correct data type and thus, potentially, improving coverage. The solver can also be used to generate data matching both the pattern and an attack payload. For example, solving both the email pattern `.*@.*\.[a-z]{2,3}`, and the classic XSS payload `<script>alert(1)</script>` results in the following solution `<script>alert(1)</script>@0.aa`.

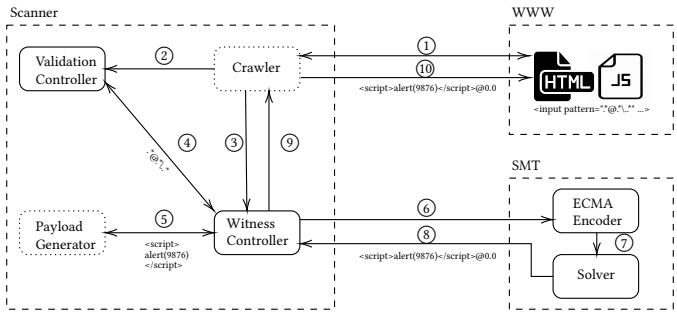


Figure C.1: The system architecture including both extended scanner and SMT solver.

C.2.1 Overview

Figure C.1 demonstrates how to extend a scanner to interact with an SMT solver. The two main components are pattern-extraction and hooking form submissions to generate valid input data. For a given target page, our extended scanner extracts all forms and analyzes the input validations. If a regex-based validation is found it communicates the regex to the string solver and submits the result. In the attack phase, the attack payload is added as an additional constraint to the solver, ensuring that the final payload is valid. We include more detailed steps in Section C.IX.

C.2.2 Motivating Example

In this section, we walk through an example where patterns are used. The scanner's crawler component requests a page as shown by ① in Figure C.1. We mark the crawler as dashed in the figure to highlight that this can be any off-the-shelf crawler. The page it crawls could, for example, validate emails using the pattern `.*@.*\..*`. In step ② the crawler sends the response, including any patterns, to the validation controller. It extracts all the patterns from the web page. This also includes dynamic interaction with the page to extract regex use in JavaScript. Before the scanner submits this form it needs to pick a witness first, which is done by calling the witness controller in step ③. The witness controller decides what type of data to send, e.g.,

a username, unique data token, XSS payload, other payload, etc. It looks up the elements it needs to submit in the validation controller in step ④. The validation controller returns the pattern, i.e. `.*@.*\..*`. The next step depends on if the scanner is in the *crawl* phase or *attack* phase.

Crawl Phase. The witness controller sends the pattern directly to the SMT in step ⑥. Inside the SMT the ECMA encoder transforms the HTML5 pattern to an SMT-LIB compliant pattern and sends it to the solver in step ⑦. The solver finds a string matching the pattern, e.g., `0@0.0`. It returns the solution to the witness controller, step ⑧, which returns it to the crawler, step ⑨, where it is submitted to the application, step ⑩.

Attack Phase. The witness controller calls the off-the-shelf payload generator to get a payload in step ⑤. The payload generator chooses a payload based on the target vulnerability. For XSS, this is usually a string like `<script>alert(1)</script>`. In step ⑥, the witness controller sends both the input element pattern and the payload to the SMT. Both are encoded and sent to the solver in step ⑦. The solver generates a valid solution to the pattern that also contains the desired payload and sends it back in ⑧. The solution `<script>alert(1)</script>@0.0` matches the pattern and contains the payload. Finally, the witness controller sends it to the crawler, step ⑨, which submits it to the web page in step ⑩.

C.2.3 Scanning

To find vulnerabilities in a web application the scanner must be able to: explore the application in a meaningful way, i.e. crawling, and methodically attack the application to find vulnerabilities.

Crawling. Classic crawling by statically parsing HTML works well for static pages but lacks many features required for scanning modern applications.

JavaScript is used by almost all web applications today. Using JavaScript, developers can add more interaction to the application, for example, by dynamically adding HTML elements like links when users press a button. The jÄk [39] scanner was the first scanner to not only execute JavaScript but also to model possible events, such as button presses.

In addition to handling dynamic client-side interactions, a scanner must also consider the dynamic server-side code. The server-side code, which is not accessible to the scanner in a black-box model, is responsible for the interactions with the database. This includes tasks like authentication, posting comments, updating user profiles, etc. This is important to handle as some actions, e.g., adding a comment can result in new parts of the application to explore. The Enemy of the State [15] scanner made strides in this area by inferring the server-side state of the scanner and thus being able to model how different actions affected the application.

While our general method of combing a scanner and string solver works for any scanner, we choose to utilize the Black Widow [18] scanner in this paper. Black Widow combines multiple methods to handle the aforementioned challenges. We

improve on this scanner by adding features that allow Black Widow to interpret and solve input validation patterns.

To find the client-side validation patterns we instrument the scanner to extract the pattern attributes from any form input element it finds. The patterns are added to the navigation graph together with the other attributes of the input element, like type and name.

Whenever the scanner needs to pick a value for an input element, it will look up the pattern for the element in the navigation graph. If a pattern is found, the SMT solver is called in real-time to find a string matching it.

Fuzzing. The most reliable method for detecting XSS is executing JavaScript and searching for the expected runtime behavior of the payload, for example showing a pop-up with the text "XSS". To further minimize false positives the payloads must be unique to each input parameter as stored payloads might be reflected in multiple places.

The Black Widow scanner uses unique payloads and dynamic injection detection already minimizing the false positives. However, the payload is limited to unique *numeric* ids. That is, the payloads have the following format `<script>xss(123456)</script>`, where `123456` will be changed for each payload. As some validation mechanisms might reject numbers, we extend Black Widow to also handle alphabetic ids.

The generated payloads should also match any validation patterns. Recall the real-world pattern `.*France` from our dataset. Here the payload must end with `France`. To generate a payload, the payload generator will use an XSS payload with a unique id, e.g. `<script>xss(123)</script>`. The string solver then tries to create a string with this payload that match the pattern. Using the pattern above, a possible solution is `<script>xss(123)</script>France` While our focus is on XSS, the same method can be used to generate SQLi payloads matching the same pattern, e.g. `'DROP TABLE;--France`

C.3 String Solving for Scanning

This section introduces the SMT component of Black Ostrich in more detail, namely, the dashed box labeled as "SMT" in Figure C.1. Black Ostrich builds on the existing open-source string solver Ostrich [10] but extends it for security scanning.

C.3.1 Overview of Ostrich

Ostrich, the "Solver" in Figure C.1, is an automata-based SMT solver specialized for string constraints. Such constraints take the form of regular expression membership assertions, and assignments of string functions (including concatenation, and all functions that can be represented as a finite-state transducer) to variables. In addition to the string operations, Ostrich has all the standard features of an SMT solver, for instance, handling of Boolean structures as well as support for other theories like integers and arrays. Given a set of assignments and assertions, Ostrich finds a model, that is, assignments of concrete strings to variables, or reports that the given formulas are inconsistent.

The whole solver is modular. Custom functions and assertions can be easily added and indeed one of the contributions of this paper is the extension of Ostrich with ECMA-style regular expression, which is covered in Section C.3.3. Before going into that, we complete the high-level overview of the SMT module by providing the translation step necessary when interfacing with web applications, that is, steps 6 and 7 in Figure C.1.

C.3.2 Translation of Validation Constraints

For scanning and fuzzing, Black Ostrich has to generate input data satisfying HTML5 form validation constraints that are specified by a web application. We focus on the attributes described in Table C.5 in Section C.VII; some further form types are less relevant from a security point of view, and they are not considered here. Our general approach is to generate data for the individual input fields of a web page one by one, which is possible because HTML5 cannot express constraints relating multiple fields so that the fields are independent.

The `type` of a field specifies the general form of data that is expected and determines how exactly data generation works. For fields with numerical input types, we can directly compute suitable input, and no constraint solving is necessary. In particular, if the input type is `number`, data has to be an integer or fractional number in decimal notation, possibly subject to side-conditions `min`, `max`, and `step`. We can directly compute a suitable input number as follows: *(i)* if `min` is present, this specified minimum value is chosen; *(ii)* otherwise, if `max` is present, then the biggest multiple of the step size is chosen that does not exceed `max`, namely $\max\{n \cdot \texttt{step} \mid n \in \mathbb{Z}, n \cdot \texttt{step} \leq \texttt{max}\}$; *(iii)* if neither `min` nor `max` is specified, the number 0 is taken. The types `color`, `range`, `date`, `datetime-local`, `month`, `time`, and `week` can be handled similarly.

For fields f of other types (e.g., `text`), an SMT formula $\phi_f[w]$ that represents all attributes that input w for f has to satisfy is constructed. This formula is sent to the SMT component, which will then check whether input data exists that fits the input field. The formula has the shape $\phi_f = \phi_{\text{type}} \wedge \phi_{\text{len}} \wedge \phi_{\text{pat}}$, and consists of a pre-defined regular expression constraint $\phi_{\text{type}} = (w \in \mathcal{L}_{\text{type}})$ that is specific to the field type, a numeric constraint $\phi_{\text{len}} = (|w| \in [l, u])$ that captures the attributes `minlength` and `maxlength` (if present), and the HTML5 `pattern` constraint $\phi_{\text{pat}} = (w \in \mathcal{L}_{\text{pat}})$ discussed in the next section. The type-specific language $\mathcal{L}_{\text{type}}$ defines all inputs that are well-formed according to the HTML standard [45]; for instance, for an input field of type `email`, we can choose the language given in Listing 1.

C.3.3 ECMAScript Regular Expressions

The `pattern` attribute enables web developers to specify further constraints on textual input using ECMAScript regular expressions [25]. Such regular expressions offer several features not present in traditional, textbook regular expressions: *(i)* anchors `^`,`$` that check for the beginning or end of a string; *(ii)* look-aheads and look-behinds, which constrain accepted strings without consuming any characters;

(iii) capture groups and back-references to the contents of those groups; *(iv)* greedy and lazy matching.

Back-references (iii) are conceptually difficult to handle, since they enable the definition of context-sensitive languages, and immediately lead to undecidability [36]. In our experiments we observed, however, that back-references are only used very infrequently within patterns; they only occur in 26 out of the unique 9 805 patterns. Compare this to look-arounds that are used 500 times and anchors that are 4044 times. Greediness (iv) of matching is not relevant for HTML patterns, as it only matters for extraction or replacement of sub-strings. We, therefore, focus on (i) and (ii).

Example 1. A regular expression commonly used as pattern for passwords is [45]:

```
^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?!.*\s).*$
```

The assertions `(?=...)` are positive look-aheads, and mandate that a password has to contain at least one digit, one lower-case letter, and one upper-case letter. The negative look-ahead `(?!...)` forbids whitespace characters.

As a second real-world example, among the patterns considered in Section C.5.1, we observed the following regular expression describing email addresses:

```
^(?=.{1,64}@)(([a-zA-Z0-9!#$%&amp;'*+-/=?^_`{|}~]+
(.[a-zA-Z0-9!#$%&amp;'*+-/=?^_`{|}~]+)*)|('.+'))@
([^-@][a-zA-Z0-9-]{1,62}.)+[a-zA-Z]{1,63}$
```

The look-ahead is in this case used to restrict the local-part to at most 64 characters.

Definitions. For ease of presentation, we adopt a mathematical notation and ignore further technicalities like Unicode alphabet, character classes, and the different escaping conventions. Let $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ be a finite set of symbols and Σ^* the set of (finite-length) strings obtained by concatenating symbols in Σ. The set of textbook regexes R is inductively defined as follows [27]:

$$\mathsf{r} ::= \emptyset \mid \varepsilon \mid \sigma \mid \mathsf{r}^* \mid \overline{\mathsf{r}} \mid \mathsf{r}_1 \cdot \mathsf{r}_2 \mid \mathsf{r}_1 + \mathsf{r}_2$$

where $\sigma \in \Sigma$, $\overline{\mathsf{r}}$ is the complement of r, * is the Kleene star operator and $\cdot$ and $+$ are the usual concatenation and alternation operators, respectively. It is also practical to define the following syntactic shortcuts:

1. $\mathsf{r}_1 \cap \mathsf{r}_2 ::= \overline{\overline{\mathsf{r}_1} + \overline{\mathsf{r}_2}}$ and
2. with slight notational abuse, $\Sigma ::= \sigma_1 + \ldots + \sigma_n$.

It is well-known that finite-state automata and regular expressions are equivalent, in the sense that they can express the same set of languages. Reasoning on them, namely checking, e.g., emptiness or inclusion, can therefore be performed using automata techniques.

On the other hand, the set of "augmented" regexes $\mathcal{R}$ include the features (i) and (ii) but they lack complementation, and they are inductively defined as follows:

$$\begin{aligned} \rho \quad ::= \quad & \emptyset \mid \varepsilon \mid \sigma \mid \rho^* \mid \rho_1 \cdot \rho_1 \mid \rho_1 + \rho_1 \mid \\ & (> \rho) \mid (\overline{>} \rho) \mid (< \rho) \mid (\overline{<} \rho) \mid \hat{} \mid \$ \end{aligned}$$

where $(> \rho)$ and $(\not> \rho)$ are the positive and negative look-ahead operators, which check if ρ matches, resp., does not match, the suffix of the string, without consuming any symbols. $(< \rho)$ and $(\not< \rho)$ are the positive and negative look-behind operators, which, analogously to the previous ones, check if ρ matches in the part of the string that has already been analyzed. Lastly, anchors ^ and $ are true only at the beginning, resp., end of the string. We refer to Appendix C.III for the formal definition of the language $L(\rho) \subseteq \Sigma^*$ described by a regex ρ.

C.3.4 Previous Results for ECMAScript Regexes

Loring et al. [36] present a symbolic execution tool for JavaScript, ExpoSE, which is able to handle also ECMAScript-compatible regular expressions excluding look-behinds. Since [36] have the goal of supporting full ECMAScript regexes, for which the language emptiness problem is undecidable, they apply an abstraction refinement loop. Initially, ECMAScript regexes are translated to SMT-LIB regular expressions (aka textbook regular expressions), which are supported by many SMT solvers; this translation is over-approximate, so that the resulting constraints might have solutions even though the original regex described an empty language. Such spurious solutions are eliminated using a refinement loop.

We can note, however, that the proposed translation from ECMAScript regexes to SMT-LIB regexes in [36] does not yield correct over-approximate constraints in many cases involving look-arounds. In particular the interaction of alternation and look-aheads, or of repetition (Kleene star) and look-aheads, is not correctly modelled, leading to an incorrect encoding of regular expressions like $((> a^* \cdot x) \cdot a)^* \cdot x$. This regex is equivalent to $a^* \cdot x$, but the translation defined in [36] interprets the regex as defining the language $\{x\}$. We conjecture that this issue is inherent in the strategy of directly translating ECMAScript regexes to SMT-LIB constraints, since a correct translation needs to handle the unboundedly many look-aheads $(> a^* \cdot x)$ caused by the outer Kleene star, and thus some notion of quantification.

C.3.5 From ECMAScript Regexes to Automata

We propose a two-layered approach to correctly handle ECMAScript regexes, implemented in Ostrich. Ostrich first attempts to translate ECMAScript regexes directly to SMT-LIB regexes (Appendix C.IV). This translation is semantics-preserving, and only causes a linear increase in size; but it is partial, and does not apply, among others, to the regex $((> a^* \cdot x) \cdot a)^* \cdot x$. Although the translation is partial, however, it still achieves high coverage of the 8 821 HTML patterns we collected. When the partial translation fails, Ostrich applies a complete approach for translating ECMAScript regexes to non-deterministic finite-state automata (NFA, used both in singular and plural); this translation can handle all features of ECMAScript regexes, with the exception of back-references. We first present a novel translation from ECMAScript regexes to two-way alternating automata (2AFA) [33], and translate 2AFA to NFA in Section C.4.

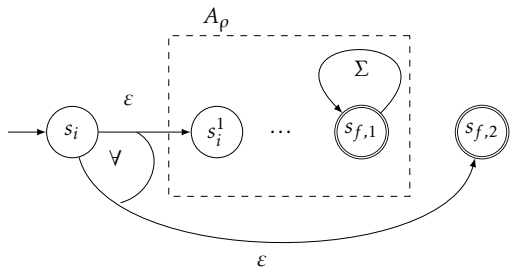


Figure C.2: Schematic representation of automaton construction recursive step for $\Diamond\rho$, where $\Diamond \in \{>, <, \geq, \leq\}$.

Automaton construction. Our automata [33] are machines that read input words. They are *two-way* in that they can move on the input both left-to-right and right-to-left, and *alternating*, meaning that they can take both existential ($\exists$) and universal ($\forall$) transitions. An $\exists$-transition corresponds to the transitions in a standard NFA: from some state, the automaton can transition to one out of multiple possible successor states. For the automaton to accept the word, it is enough if one such execution is successful. Conversely, $\forall$-transitions fork the execution to a set of paths that should *all* be successful. For both kinds of transitions, the automaton also specifies if it is moving forward or backward. It is well known that 2AFA has the same expressive power as standard NFA, although being exponentially more succinct, and indeed the former can be simulated by the latter [7, 22, 29]. These algorithms, however, besides having exponential complexity, are also quite intricate and have never been implemented in the context of SMT solvers, to the best of our knowledge. We, therefore, introduce a new version of 2AFA, which we call 2AFA_{SMT} with the following features: *(i)* their semantics is closer to the semantics of ECMAScript regex, thus enabling a more direct representation of those and *(ii)* they allow for a simple and practically efficient translation to NFA. The main difference between traditional 2AFA and 2AFA_{SMT} is on the way transitions are specified. The former reads the character they are currently analyzing and then moves either forward or backward positioning themselves on the respective character, while the latter sits in-between characters, and they can either read the preceding one and move backward, or the succeeding one and move forward. This is obtained by having two different kind of transitions, the backward $\delta^<$ transitions and the forward $\delta^>$ ones.

Definition C.1 (2AFA_{SMT})**.** A *two-way alternating automaton* is a tuple $(\Sigma, S, s_0, F^<, F^>, \delta^>_\exists, \delta^<_\exists, \delta^>_\forall, \delta^<_\forall, \varepsilon_\exists, \varepsilon_\forall)$ where:

1. Σ is an alphabet of symbols;
2. S is a finite set of states;
3. $s_0 \in S$ is an initial state;
4. $F^<, F^> \subseteq S$ are disjoint sets of final states;
5. $\delta^>_\exists, \delta^<_\exists : S \times \Sigma \dashrightarrow \wp(S)$ are the partial existential transition functions;

6. $\delta^{>}_{\forall}, \delta^{<}_{\forall} : S \times \Sigma \dashrightarrow \wp(S)$ are the partial universal transition functions;
7. $\varepsilon_{\exists} : S \dashrightarrow \wp(S)$ is the partial ε-existential transition function;
8. $\varepsilon_{\forall} : S \dashrightarrow \wp(S)$ is the partial ε-universal transition function.

and $\wp(S)$ is the powerset of S. We require that for every $s \in S, \sigma \in \Sigma$ one of the δ or ε transitions is defined.

Next, we define the semantics of an automaton, namely the set of words it accepts.

Definition C.2 (2AFA$_{SMT}$ run)**.** Let $w = w_0 w_1 \dots w_n$ be a word in Σ^* of length $\ell(w) = n+1$, A be 2AFA$_{SMT}$ and $i \in \mathbb{N}$. A run π of A on w is a finite sequence of elements in $\wp(S \times \mathbb{N})$, called *configurations*, and it is defined inductively: $\pi_0 := \{(s_0, 0)\}$ and for any π_j with $j \in \mathbb{N}$ we build (one of the possible successor configurations) π_{j+1} as follows. Pick $(s,i) \in \pi_j$, then: $\pi_{j+1} := \pi_j \setminus \{(s,i)\} \cup T$ where T is one of the following:

1. $\{(s', i+1)\}$ if $i < \ell(w)$ and $s' \in \delta^{>}_{\exists}(s, w(i))$;
2. $\{(s', i-1)\}$ if $i > 0$ and $s' \in \delta^{<}_{\exists}(s, w(i-1))$;
3. $\{(s'_1, i+1), \dots, (s'_n, i+1)\}$ if $i < \ell(w)$ and $\delta^{>}_{\forall}(s, w(i)) = \{s'_1, \dots, s'_n\}$;
4. $\{(s'_1, i-1), \dots, (s'_n, i-1)\}$ if $i > 0$ and $\delta^{<}_{\forall}(s, w(i)) = \{s'_1, \dots, s'_n\}$;
5. $\{(s', i)\}$ if $s' \in \varepsilon_{\exists}(s)$;
6. $\{(s'_1, i), \dots, (s'_n, i)\}$ if $\varepsilon_{\forall}(s) = \{s'_1, \dots, s'_n\}$.

We say that automaton A *accepts* w if there exists a run π of A over w in which the last configuration is accepting, that is: for each (s,i) we have either $s \in F^{<}$ and $i = 0$ or $s \in F^{>}$ and $i = \ell(w)$. Intuitively, a pair (state, index) (s,i) means that the automaton is in state s and in-between the $i-1$-th and i-th characters of w. Being alternating, we might have more than one pair at any moment, as the automaton is scanning multiple parts of the word at the same time. We start from initial state s_0 at the beginning of the word (index 0) and at each step a pair is picked and a transition is performed: if such transition is existential, then the current state is updated with one of the successor states; if it is universal, all the successor states are added to the current run. Also the index is updated depending on if the transition is moving backward $<$ or forward $>$.

Translation of augmented regexes. The procedure we propose recursively constructs a 2AFA A_ρ for each augmented regex $\rho \in \mathcal{R}$. Compared to similar constructions in the literature [27], ours adds cases for handling look-arounds and anchors. We notice that the latter can be seen as shortcuts: it is indeed easy to prove that ^ is equivalent to $(\overleftarrow{<}\Sigma)$ and \$ is equivalent to $(\overrightarrow{>}\Sigma)$. We discuss the main cases of the translation in this section and refer the reader to Appendix C.III for further details.

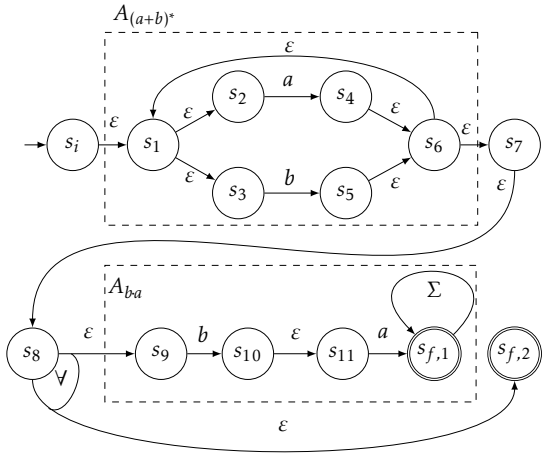


Figure C.3: Graphical representation of the automaton for the regex $(a+b)^* \cdot (< b \cdot a)$. Unless marked with $\forall$, transitions are existential.

Example 2. Consider the regex $(a+b)^* \cdot (< a \cdot b)$. The regex is translated to the automaton in Figure C.3, and illustrates the translation of concatenation, the + operator, and a look-behind. When running on $w = abbbab$, a successful execution sees the sub-automaton $A_{(a+b)^*}$ matching the whole word and ending in state s_8. From there, the execution forks: one path directly accepts while the other goes through sub-automaton $A_{a \cdot b}$ which starts scanning backward. It first reads b and then a, (which indeed matches $(< a \cdot b)$) thus ending up in the sink accepting state s_{f_1}. Since both paths are in a final state, word w is accepted.

Intuitively, the automaton translation works as follows (see Appendix C.III for more details and figures). The atomic cases of ε, σ accept after seeing ε or σ, respectively. Automaton for $\emptyset$ never accepts. The automaton for alternation forks the execution with $\exists$-transition into two paths, each attempting to match a subexpression. Automaton for concatenation connects with an ε-transition the automata for the subexpression and concerning the Kleene star, an initial $\exists$-transition forks the execution two paths, one directly accepts, the other matches repetitions of ρ by adding an ε-cycle from the final state to the initial state of (sub-)automaton for ρ. The novel case of lookahead $(> \rho)$ builds the automata schematized in Figure C.2, where the dashed box is the automaton for ρ, double-circled states are final (more precisely, $s_{f,1} \in F^<$ and $s_{f,2} \in f_>$) and the outgoing transitions from s_i are $\forall$-transitions (recursively, the final state $s_{f,2}$ will be possibly expanded into an automaton that recognizes the remaining part of the regex). Notice the initial $\forall$-transition which forks the execution in two paths that should both accept. When a look-behind is encountered, the same idea holds, but the automaton inside the dashed box scans the word backward, hence the necessity of a two-way automaton. We reverse the regex inside the look-behind, hinging on the fact that scanning the reverse of a word w backward from end to start is equivalent to scan w forward from the beginning.

Lastly, the negated look-arounds are handled by complementing the resulting automata [21].

Theorem 1. *Let $\rho \in \mathcal{R}$, and A_ρ be the two-way alternating automaton constructed from ρ. Automaton A_ρ accepts a word $w \in \Sigma^*$ if and only if $w \in L(\rho)$.*

We refer the reader to Appendix C.III for the proof. We also remark that:

Lemma C.1. Building A_ρ for $\rho \in \mathcal{R}$ takes linear time in the size of ρ.

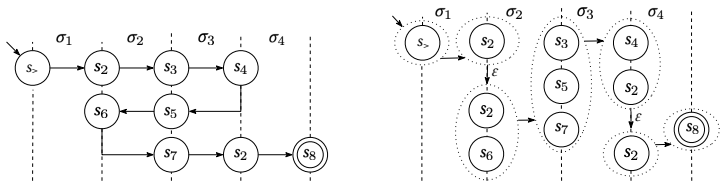


Figure C.4: An example of a run of a S-2AFA$_{SMT}$ on the left and the corresponding run of the simulating NFA on the right.

C.4 From 2AFA$_{SMT}$ to NFA

We now define a translation from 2AFA$_{SMT}$ to a standard non-deterministic finite-state automaton (NFA). An NFA is an automaton where the transition function is non-deterministic and always moves left-to-right, but which might also have ε-transitions. More precisely, a NFA is a tuple $(\Sigma, Q, q_0, Q_f, \delta, \varepsilon)$ where $q_0 \in Q$ is the initial state, $Q_f \subseteq Q$ the set of final states and $\delta : Q \times \Sigma \dashrightarrow \wp(Q)$ and $\varepsilon : Q \dashrightarrow \wp(Q)$ are non-deterministic transitions. The semantics of NFA is defined similarly to that of 2AFA$_{SMT}$. A run π of an NFA over a word w is finite sequence of configurations $Q \times \mathbb{N}$ defined inductively: $\pi_0 = (q_0, 0)$ and for any $\pi_j = (q, i)$ with $j \in \mathbb{N}$ we have (one of the possible successors) $\pi_{j+1} = (q', i')$ if and only if either *(i)* $q' \in \varepsilon(q)$ and $i = i'$; or *(ii)* $i < \ell(w)$, $q' \in \delta(q, w(i))$ and $i' = i + 1$.

Our translation to NFA requires a further assumption about the considered 2AFA$_{SMT}$: we say that a 2AFA$_{SMT}$ A is *non-cycling* if, for every word w, the set of (accepting or non-accepting) runs according to Def. C.2 on w is finite. Existing methods for transforming non-cycling 2AFA into an NFA [7, 22, 29] are inspired by the original Shepherdson's construction [43] for eliminating the bidirectionality, and on the powerset construction for removing the universal transitions. Here we propose a translation which is also inspired by previous results and eliminates bidirectonality and universal alternation with a one-step powerset construction, which is intuitive yet efficient in practice. Such a construction works only for non-cycling 2AFA$_{SMT}$, which is indeed the case of our 2AFA$_{SMT}$ built from the augmented regex in Section C.3.

The intuition behind our approach is to categorize states based on the direction, left-to-right > or right-to-left <, from which they can be reached and on the direction

they can be left. We denote the former with a superscript and the latter with a subscript. For example, a state belonging to set $S^{>}_{<}$ means that it can be reached only with a left-to-right transition (> in the superscript) and can be left with a right-to-left transitions (< in the subscript). Such a categorization is required to define the simulating NFA.

Definition C.3 (S-2AFA$_{SMT}$)**.** A simplified 2AFA$_{SMT}$, denoted S-2AFA$_{SMT}$, is a tuple $(\Sigma, S^{>}_{>}, S^{<}_{<}, S^{>}_{<}, S^{<}_{>}, S^{>}, s_{>}, F^{>}, \delta^{>}_{\exists}, \delta^{<}_{\exists}, \delta^{>}_{\forall}, \delta^{<}_{\forall})$:

1. the set of states $S^{>}_{>}, S^{<}_{<}, S^{>}_{<}, S^{<}_{>}, S^{>}, \{s_{>}\}$ are pairwise disjoint, and we denote with S their union;

2. $s_{>}$ is the initial state, which does not have incoming transitions and has only left-to-right outgoing transitions;

3. $S^{>}$ is the set of sink states that only have incoming left-to-right transitions, namely for each $s \in S^{>}$, $q \in S$, $\sigma \in \Sigma$ and $* \in \{\exists, \forall\}$ we have that: $s \notin \delta^{<}_{*}(q, \sigma)$ and $\delta^{<}_{*}(s, \sigma)$ and $\delta^{>}_{*}(s, \sigma)$ are undefined;

4. $F^{>} = S^{>}$ are final states;

5. for each state $s \in S^{>}_{>}$, $q \in S$, $\sigma \in \Sigma$ and $* \in \{\exists, \forall\}$ we have that: $s \notin \delta^{<}_{*}(q, \sigma)$ and $\delta^{<}_{*}(s, \sigma)$ is undefined. Analogous definitions hold for $S^{>}_{<}, S^{<}_{>}, S^{<}_{<}$.

6. The transition functions are as in Definition C.1.

We notice that any 2AFA$_{SMT}$ can be transformed into a S-2AFA$_{SMT}$, and refer to Appendix C.III for the procedure.

Next, we show how to build a NFA (with ε transitions) that is equivalent to a S-2AFA$_{SMT}$. Intuitively, states of the NFA are subset of states of the S-2AFA$_{SMT}$, which we call macro-states henceforth, and transitions are defined by suitably considering each category of states. The left hand-side of Figure C.4 pictures a run of a S-2AFA$_{SMT}$ on word $w = \sigma_1\sigma_2\sigma_3\sigma_4$ where automaton states are in-between characters, on the dashed vertical lines. Starting from the state $s_{>}$, the automaton reads three characters moving right ($s_2, s_3 \in S^{>}_{>}$) and lands in state $s_4 \in S^{>}_{<}$; then it moves backward on $s^5 \in S^{<}_{<}$ and ends up in $s_6 \in S^{<}_{>}$, and then finally moves forward to the end of the word accepting in $s_8 \in S^{>}$. The simulating NFA scans instead the word left-to-right only once, essentially guessing at each step the possible (forward and backward) computations of the S-2AFA$_{SMT}$, as depicted on the right hand-side of Figure C.4. Dashed circles represent the macro-states of the NFA.

Definition C.4 (Simulating NFA)**.** Let $(\Sigma, S^{>}_{>}, S^{<}_{<}, S^{>}_{<}, S^{<}_{>}, S^{>}, s_{>}, F^{>}, \delta^{>}_{\exists}, \delta^{<}_{\exists}, \delta^{>}_{\forall}, \delta^{<}_{\forall})$ be a S-2AFA$_{SMT}$. The equivalent NFA is: $(\Sigma, \wp(S), \{s_{>}\}, \{F \mid F \subseteq F^{>}\}, \delta, \varepsilon)$ where the following restrictions holds. For every $Q, Q' \in \wp(S)$, we have $Q' \in \varepsilon(Q)$ if and only if either:

1. $Q' = Q \cup \{s\}$ and $s \in S^{<}_{>}$ or

2. $Q' = Q \setminus \{s\}$ and $s \in S^{>}_{<}$.

Also, for each $Q, Q' \in \wp(S)$ and $\sigma \in \Sigma$, we have $Q' \in \delta(Q, \sigma)$ if and only if the conjunction of the following holds:

1. $Q \cap F^{>} = \emptyset$ and $Q \cap S^{>}_{<} = \emptyset$;
2. $Q' \cap \{s_{>}\} = \emptyset$ and $Q' \cap S^{<}_{>} = \emptyset$;
3. *right-successors:* for all $s \in Q$, if $s \in S^{>}_{>} \cup S^{<}_{>} \cup \{s_{>}\}$, then $Q' \cap \delta^{>}_{\exists}(s, \sigma) \neq \emptyset$ or $\delta^{>}_{\forall}(s, \sigma) \subseteq Q'$;
4. *left-successors:* for all $s' \in Q'$, if $s' \in S^{<}_{<} \cup S^{>}_{<}$, then we have $\delta^{<}_{\exists}(s', \sigma) \cap Q \neq \emptyset$ or $\delta^{<}_{\forall}(s', \sigma) \subseteq Q$;
5. *right-predecessors:* for all $s \in Q$, if $s \in S^{<}_{>} \cup S^{<}_{<}$, then there exists $s' \in Q'$ such that $s \in \delta^{<}_{\exists}(s', \sigma)$ or $s \in \delta^{<}_{\forall}(s', \sigma) \subseteq Q$;
6. *left-predecessors:* for all $s' \in Q'$, if $s' \in S^{>}_{<} \cup S^{>}_{>} \cup F^{>}$, then there is $s \in Q$ such that $s' \in \delta^{>}_{\exists}(s, \sigma)$ or $s' \in \delta^{>}_{\forall}(s, \sigma) \subseteq Q'$.

The conditions on the transition functions follow from the shape of the S-2AFA$_{SMT}$ runs. For example, referring to Figure C.4, we have that states in $S^{<}_{>}$, such as s_6, can "appear" in a macro-state, Q_2 in this case, thanks to ε transitions as dictated by the first bullet in Definition C.4 (analogously, $S^{>}_{<}$, such as s_4, can disappear). However, if they appear, then a state they come from should exist from the right (case (5)) as well as one where they go to, again to the right (case (3)). Similar conditions hold for $S^{>}_{<}$ states, while for $S^{>}_{>}$ and $S^{<}_{<}$ states we simply require the existence of successor(s) and a predecessor on the right or on the left, respectively.

Theorem 2. *For any word w on Σ, w is accepted by a non-cycling S-2AFA$_{SMT}$ iff w is accepted by its simulating NFA.*

Proof. "$\Leftarrow$" Fix a word w, and let $\pi = ((Q_0, 0), (Q_1, i_1), (Q_2, i_2), \ldots, (Q_n, \ell(w)))$ be the accepting run of the simulating NFA. Let $P = \{(s, i) \mid (Q_i, i) \in \pi \text{ and } s \in Q_i\}$ be the set of 2AFA$_{SMT}$ state/index pairs occurring in the run. Consider then the graph $G = (P, R)$ with:

$$R = \begin{array}{l} \{((s,i),(t,i+1)) \mid t \in \delta^{>}_{\exists}(s, w(i)) \cup \delta^{>}_{\forall}(s, w(i))\} \cup \\ \{((s,i),(t,i-1)) \mid t \in \delta^{<}_{\exists}(s, w(i-1)) \cup \delta^{<}_{\forall}(s, w(i-1))\} \end{array} \tag{C.1}$$

G is acyclic, because the considered S-2AFA$_{SMT}$ is non-cycling. We prove that, for every $(s, i) \in P$, there is an accepting S-2AFA$_{SMT}$ run π on w with the initial configuration $\pi_0 = \{(s, i)\}$; this is proven by well-founded induction on the length of the longest path in G starting from (s, i). If (s, i) has no outgoing edges, then $i = \ell(w)$ and s must be accepting, and $\pi = (\{(s, i)\})$ is an accepting run.

Assume then that (s, i) has outgoing edges, and suppose the first case of (C.1) applies and $s \in S^{>}_{>} \cup S^{<}_{>} \cup \{s_{>}\}$ (the second case is similar). By Def. C.4, there is an $s' \in \delta^{>}_{\exists}(s, w_i)$ such that $(s', i+1) \in P$, or $(s', i+1) \in P$ for all $s' \in \delta^{>}_{\forall}(s, w_i)$. In both cases, by induction for each such s' an accepting run with initial configuration $\{(s', i+1)\}$ exists, which can be combined and extended to obtain an accepting run with initial state $\{(s, i)\}$.

"⇒" Proven in a similar way, this time by considering the graph G generated by an accepting S-2AFA$_{SMT}$ run on w and showing that at each step of a run, the set of states composing the run are included in macro-states Q of an NFA run on w. ■

C.5 Coverage and Vulnerability Study

We evaluate our approach by performing a large-scale scan of patterns used on the web. We explain how we gather the patterns in Section C.5.1. We add these patterns to a testbed on which we can compare our approach with other state-of-the-art scanners. Details about the testbed can be found in Section C.5.2. Design choices for the implementation of Black Ostrich are presented in Section C.5.3.

C.5.1 Gather Data

To compare our method with other scanners we create a testbed to test both the crawling capabilities of scanners and their payload generation capabilities. We do this by extracting forms using HTML5 patterns from real-world websites.

To find real-world client-side validation, we use data from the Common Crawl data set [11] of websites. The Common Crawl data set is a public collection of HTML documents found while crawling the public web. From Common Crawl we extract all archives of the July 2021 index collection (`CC-MAIN-2021-31`), containing 3,15 billion pages or 360 TiB uncompressed content. The process took about two weeks. To avoid over-collecting, we deduplicate incoming validation patterns per archive (`.warc`) file, meaning that e.g. recurring input validation patterns in a page header are only reported once per archive. This strategy is based on an assumption that pattern duplication exhibits locality with repeated patterns occurring within the same archive. For each page crawled we extract all the HTML patterns and create a database along with some of their contexts such as other attributes of the element and the URL at which they were initially crawled.

In addition to Common Crawl, we also crawl the top 100K domains from Tranco [34] to include patterns from popular websites. For each domain, we pick five random links and search all pages for forms with HTML5 patterns.

In total, we extract 9 805 unique patterns. We further remove any patterns that cause a syntax error in either Node, Firefox, or PHP. Inconsistent patterns, where a solution is accepted in JavaScript but not PHP or vice versa, and patterns reported as unsatisfiable by both Black Ostrich and ExpoSE are also removed. This results in 8 821 valid patterns that we use for the testbed, and share publicly [3].

The most common pattern by far is for checking email addresses. This is interesting as `type="email"` already supports similar validation. Other popular ones are the semantically equivalent patterns `[0-9]*` and `\d*`. Usually corresponding input elements for *quantities*. In general, the complexity spans from simple and short to long and complex. The average length of the patterns is 39 characters but there are 453 longer than 100 characters and the longest pattern is 29 059. There are also 500 patterns using look-arounds and 4044 patterns using anchors.

C.5.2 Testbed

To avoid attacking live websites we recreate the same input elements in a testbed. Using the real-world patterns we design this testbed with one page per unique pattern. Each page in the testbed consists of a form with a single input element containing a pattern from the database. We also include the most common name and type for each pattern as some scanners use this information to decide what value to send. A template of the testbed page used for each pattern is shown in Section C.VI in Section C.VI.

The same pattern is also checked server-side to ensure scanners do not simply ignore the client-side check. We also check the input type validation for *email* and *URL* server-side. We show this server-side code in Section C.VI in Section C.VI. Note that HTML5 patterns match on the entire string and not just part of it. Therefore we surround the pattern with `/^(?:` and `$)`, as explained in the HTML5 standard [45]. If the scanner sends a valid input the server will reflect this input, allowing for XSS, which should be detected by the scanners. Finally, we run each scanner on the testbed and record both if it passes the pattern, and if it reports the XSS vulnerability.

C.5.3 Implementation

In this section, we present the implementation details for our validation-aware scanner. We implement our approach [3] by synergizing and improving the open-source web application scanner Black Widow [18] and solver Ostrich [10]. Section C.5.3.1 presents the modifications to the scanner, and Section C.5.3.2 presents the modifications to the solver.

C.5.3.1 Scanner module.

We make two major modifications, one to the data extraction and one to the witness selection.

We update the navigation model in the crawler component to allow for modeling of the new pattern attributes. During the crawling phase, we save all the patterns the scanner finds together with their respective input elements. To dynamically detect regex-based JavaScript validation we proxy the `test` method of the `window.RegExp` object. Then we input unique tokens on all input fields and save any regex where `test` is applied on our input.

Once the scanner is ready to submit a value to a form we retrieve the corresponding pattern from the navigation model and send it to the solver, which generates a matching value for the pattern. When the scanner is in attack mode then the payload is also sent to the solver as an additional constraint.

C.5.3.2 Solver module.

As SMT solver in Black Ostrich, we apply an extended version of the open-source solver Ostrich. The difference to the standard version of Ostrich is the handling of ECMA regular expressions (Section C.3 and Section C.4). This functionality was integrated by extending the Ostrich SMT-LIB interface [5], adding a new function

`re.from_ecma2020` that converts a string to a regular expression. The translation of 2AFA_{SMT} to NFA is implemented through the expansion in Def. C.4; as an optimization, our implementation only generates reachable NFA states to mitigate the possible exponential growth caused by the expansion.

C.5.4 Comparison of Ostrich and ExpoSE

Loring et al. [36] present ExpoSE, a symbolic execution tool for JavaScript. While ExpoSE is not a string solver, it contains one that could potentially be combined with a web scanner. To compare, we run both ExpoSE and Ostrich on all valid patterns and record their coverage and execution time. Since ExpoSE uses symbolic execution we need to create a small JavaScript file with the pattern. The exact JavaScript file is in Section C.VIII. To avoid getting stuck on any particular pattern we use a timeout of ten seconds.

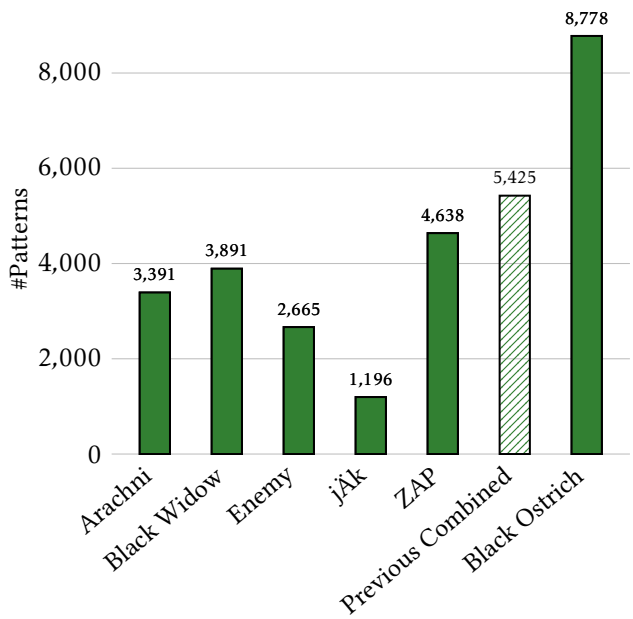


Figure C.5: Number of patterns passed by scanners.

C.6 Results

In this section, we present the results of our empirical study. Section C.6.1 presents the results from our testbed. Finally, In Section C.6.2 we analyze the results and present qualitative insights into the results.

C.6.1 Black-box Scanning

We divide the testbed results into pattern coverage and XSS vulnerability detection.

C.6.1.1 Coverage.

In total our scanner solves 8 778 patterns out of the total 8821, resulting in a coverage of 99%. In comparison, the other scanners have an average coverage of 36%, ranging from jÄk solving 1 196 patterns (14%) to ZAP solving 4 638 patterns (53%).

The coverage results are presented in Figure C.5, which shows that our method can pass many more patterns compared to the combined efforts of previous approaches. An example of a group of patterns only we find are patterns tightly bound in length, like `\d{16}`.

To compare the coverage between scanners we create a heatmap, shown in Table C.1, where each cell contains the number of patterns that the scanner on the row finds compared to the scanner on the column. For example, on the Arachnirow in the Black Ostrich column, we see that Arachni solves 8 patterns that Black Ostrich misses. Conversely, the Black Ostrich row and Arachni column show that Black Ostrich solves 5 395 patterns that Arachni misses. In this sense, neither of those scanners is strictly better than another at solving patterns.

We also see that the no scanner performs strictly better than another scanner in terms of coverage, as all zeroes lie on the diagonal. The Black Ostrich column of the heatmap highlights that our scanner is unable to solve some patterns that other scanners can handle. These cases are explored further in Section C.6.2.

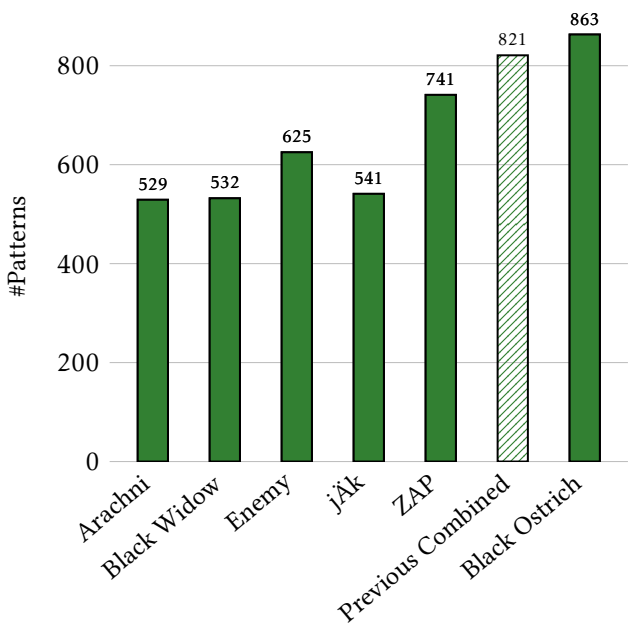


Figure C.6: Number of vulnerable patterns found by scanners.

C.6.1.2 Vulnerabilities.

Figure C.6 shows how many patterns the scanner reports are XSS vulnerable. As the data shows, our method outperforms the other scanners in terms of how well it can submit payloads that match the pattern. Compared to the average of the other scanners we improve vulnerability detection by 45%. The patterns passed by other scanners are usually simpler, like `.{7,}`, which allows any payload that is at least seven characters long. This explains the plateau at around 535 in Figure C.6. We analyze in more detail why Enemy of the State and ZAP can break this plateau in Section C.6.2. In short, they are more lenient about what constitutes proof of XSS. This results in many false positives. Our approach outperforms the others in cases with stricter formatting requirements. A simple example is email patterns that require the at-sign and period, like `.+@.+[.].+`. Or requirements of specific strings, like "France" in the pattern `.*France`.

As can be seen on the Black Ostrich row in the heatmap in Table C.2, our scanner has a strong matchup against all the other scanners when comparing found vulnerabilities. Interesting to note is a total of 98 vulnerabilities we miss, that other scanners find. We discuss these cases, and additional cases we miss, more closely in the analysis in Section C.6.2

C.6.2 Analysis

In this section, we explore some interesting parts of the results. We split the analysis into phenomena regarding the coverage, in Section C.6.2.1, and vulnerabilities, in Section C.6.2.2.

C.6.2.1 Coverage.

To better understand what can be improved we analyze the cases we miss that other scanners find. We also explore the patterns no scanner solves to detect any false negatives.

What we miss. As Table C.1 shows, there are cases where other scanners solve patterns that we are not able to solve. In total there are 15 cases where another scanner solves a pattern that we do not. These are complex patterns that have relatively easy solutions. A scanner-related problem is a pattern where the first solution is the DEL character (0x7F), which can not be typed into the text field by our scanner. To improve coverage in these cases we need to ensure the solutions are printable and improve the underlying scanner to handle submission of unprintable values.

C.6.2.2 Vulnerabilities.

Notable for the vulnerability results is that Enemy of the State and ZAP find more vulnerabilities compared to the other scanners. We also explain why jÄk has worse coverage performance yet similar vulnerability performance compared to the other scanners.

Table C.1: Unique coverage found between scanners

	Arachni	Black Ostrich	Black Widow	Enemy	JÄk	ZAP
Arachni	0	8	962	1,625	2,201	641
Black Ostrich	5,395	0	4,889	6,121	7,584	4,151
Black Widow	1,462	2	0	1,449	2,774	184
Enemy	899	8	223	0	1,506	64
JÄk	6	2	79	37	0	19
ZAP	1,888	11	931	2,037	3,461	0

Compared to Enemy of the State and ZAP. Both Enemy of the State and ZAP perform better than the other scanners we test. The reason for this is not that they use advanced string solving, but rather a different proof of XSS. This allows them to use *shorter* payloads. For example, for the pattern `.{0,20}`, which allows a maximum of 20 characters, a normal XSS payload, e.g. `<script>alert(1)</script>`, is too long at 25 characters. In comparison, Enemy of the State uses the 19 characters long string `'';!--"<0cy1>=&{()}` and ZAP uses `javascript:alert(1)`. We see these as false positives and therefore do not accept this in Black Ostrich. However, we still add support for detecting tag-injections, making it easy for developers to enable it.

jÄk's coverage and vulnerability detection. jÄk's performance is interesting as the coverage is significantly worse compared to the other scanners, yet the number of found vulnerabilities is on par with the others. This is because jÄk only sends attack payloads to the form. As such, jÄk's coverage will match the vulnerabilities they find plus any pattern accepting empty strings. This differs from scanners that also try benign values for the input elements.

C.6.3 Results of Black Ostrich vs. ExpoSE

From the 9 805 patterns, we remove broken, invalid and unsatisfiable patterns to arrive at a total of 8 821. Our experiment shows that Black Ostrich can solve 8 820 while ExpoSE solves 7 189, an improvement of 23%. This is slightly better than the testbed results as it does not impose the extra constraint of printable characters. Comparing the time performance, Black Ostrich takes on average 1.35 seconds and ExpoSE takes 4.11. Also important to note is that ExpoSE time-outs on 158 patterns while Black Ostrich only time-outs on 1 pattern.

One example of where ExpoSE fails, but Black Ostrich does not, is a large group of 358 patterns using look-arounds; one such pattern is the recommended regex for passwords shown in Example 1. In general, the patterns ExpoSE fail on are also longer, with an average length of 62 characters compared to 26 for solved patterns. Similarly for patterns that timeout, the average length is 156 vs. 30 for patterns solved in time.

Table C.2: Unique vulnerabilities found between scanners

	Arachni	Black Ostrich	Black Widow	Enemy	JÄk	ZAP
Arachni	0	15	101	86	113	59
Black Ostrich	349	0	341	280	330	206
Black Widow	104	10	0	49	97	12
Enemy	182	42	142	0	152	16
JÄk	125	8	106	68	0	28
ZAP	271	84	221	132	228	0

C.7 Patterns in Open-Source Applications

To further explore the prevalence of patterns in the wild we perform a study on the usage of patterns in open-source GitHub projects. We download the 978 best matching projects from GitHub's "web-application" topic [23]. Note that many projects here are not strictly web applications, but also projects such as Arachni, which is a web application scanner. Searching for `" pattern="` we extract 195 files from 72 projects. We manually analyze these files to determine if they are HTML patterns, as this is not always the case. For example, we found non-HMTL patterns in SVG files. We also remove projects that are lacking enough information to run locally, for example, projects missing database schemas. After filtering, we acquire three usable projects with HTML patterns. We run these and inspect them manually in the browser to ensure the patterns are used.

ALEX. The ALEX project is a great example of a web application that validates the pattern both on the client-side and server-side. To create a new project in ALEX a URL is required, and the URL must match `^https?://.*?`.

Similar to previous studies [39] we compare the number of URLs the scanner visits, excluding URLs to static files. We also ignore query parameters in the URL and collapse the ID number in the path. For example, the `projects/1/users` and `projects/2/users` execute similar code, but for different projects.

The results show that Black Ostrich can find 25 different path-level URLs while Black Widow only found nine, resulting in a 178% increase in URL-level coverage. Furthermore, Black Ostrich found all the nine Black Widow found. Neither Arachni, Enemy of the State, jÄknor ZAPmanaged to pass the login, resulting in just one URL. The authentication is extra difficult as it uses a cross-domain token service to manage authentication and not simple cookies. The login form is also dynamically generated making it impossible to specify credentials to these scanners.

By source code analysis, we determine that being able to solve the pattern needed to create projects is the key factor to achieve higher crawling coverage in this case.

Helping Hands. The main challenge in this web application is registering a user. The registration form uses patterns to validate phone numbers, among other things. Here we do not provide credentials to test their ability to register.

In this case, all scanners except Enemy of the State found the registration form. From here, only Black Ostrich was able to correctly solve all the patterns needed for registration and authentication.

Opera DNS UI. This application uses many challenging patterns in its forms relating to DNS records, like a pattern[1] for IPv4 addresses. As these are checked server-side too, only a valid IP address will be accepted in this case. As Enemy of the State does not support *basic access authentication* it could not log in. It would still fail to submit the form as the submission is triggered by JavaScript. The other scanners can log in and find the form. As the form's method is *post*, jÄkdoes not interact with it. Both ZAPand Arachnimanages to submit, but not valid data, and thus rejected by the server. Only Black Ostrich can submit a valid record.

C.8 Related Work

SMT. String constraints solvers have flourished in recent years [2]. The two main paradigms for solving string constraints are SMT and constraint programming. Many SMT solvers have decision procedures for handling string constraints among which: Z3 [13], Z3-str/2/3/4 [46], S3/p/# [44]; CVC4 [35], Norn [1], Sloth [26], and Ostrich [10]. They rely on automata-based techniques or algebraic results for strings or reduce the problem to other well-known theories, such as integers or bit-vectors. We believe our solver is the first to directly handle ECMAScript regular expressions. A comparison with the symbolic execution tool ExpoSE, which includes support for ECMAScript regexes, is provided in Section C.3.4.

2AFA. The equivalence between two-way and one-way automata has been originally proven by Rabin, but a most modern straightforward proof is given by Shepherdson in [43]. Alternation has been introduced only later, in the seminal work in the '70s [9]. Since then, the study of the combination of the two has been scattered, and we refer to [30] for a thorough survey. Although a first proof of equivalence between 2AFA and NFA appeared in [33], Birget [7] is the first to provide a comprehensive study of different kinds of finite automata, as well as a translation from (non-cycling) 2AFA to NFA which is done in several steps and make uses of homomorphisms between alphabets. More recent translations appear in [22], which also works for cycling 2AFA. The above are theoretical construction, usually oriented to complexity theory, and to the best of our knowledge they have not been implemented in practice.

Web scanning. Web scanning is a research topic actively being explored. The reason behind this is that there are still many open challenges in web scanning, both for improving crawling and for improving vulnerability detection. In this study, we compare our implementation with other state-of-the-art scanners [15, 38, 39, 41]. Outside of our empirical study, there are also many other scanners [4, 16, 17, 20, 24, 28, 40] that made significant improvements in the field. A related study by Fonseca

[1] ((25[0-5]|(2[0-4]|10,1[0-9])0,1[0-9])̇3,3(25[0-5]|(2[0-4]|10,1[0-9])0,1[0-9])

et al. [19] shows that many security patches in web applications update vulnerable regexes, further motivating the need for validation-aware web scanning.

While the goal of improving vulnerability detection has been common for previous approaches, the areas of scanning they improve vary. For example, jÄk [39], Enemy of the State [15], LigRE [16], and Black Widow [18] focuses mainly on improving the crawling aspect of scanning, while using common payloads and fuzzing techniques. jÄk improved crawling by modeling JavaScript events in a novel way leading to deeper crawls and a higher detection rate of vulnerabilities. Enemy of the State achieved similar improvements by instead inferring the server-side state, thus being able to handle more complex workflows. Black Widow improves crawling by combining key features from previous methods, including navigation modeling, traversing, and inter-state dependency analysis. Although we build our scanner on top of Black Widow, neither of these approaches covers the orthogonal aspect of handling the validation patterns supplied by web applications.

In addition to improving crawling, the attack phase can also be improved to achieve better vulnerability detection rates. Both KameleonFuzz [17] and sqlmap [20] are examples of scanners that focus more on payload selection and fuzzing techniques to improve detection rate. KameleonFuzzdynamically mutates the XSS payloads based on the reflected value to iteratively update the payload until an attack is successful. While this has the potential of solving patterns, it is probabilistic and likely fails on very specific patterns. For example, one pattern only we could exploit, was `.*France`. Finding inputs with this specific string using mutations seems highly unlikely. While we focused on XSS in this study, solving patterns is important for finding other vulnerabilities such as SQL injections too. sqlmap does not consider patterns when fuzzing, instead, they rely on a large table with payloads that use different escaping techniques. This too would fail on the vulnerable "France" example. To overcome this our scanner, Black Ostrich, also uses SMT to generate the payloads. This means that we can combine common attack payloads, like `<script>alert(1)</script>` with patterns like `.*France` to generate success attack inputs like `<script>alert(1)</script>France`.

C.9 Conclusions

We have presented Black Ostrich, a principled approach that leverages string-based constraint solving for deep crawling. We improve state-of-the-art string solving by extending the open-source solver Ostrich with native support for ECMA regular expressions. To handle the commonly occurring anchors and look-arounds in patterns on the web, we propose a new version of two-way alternating finite-state automata, named 2AFA_{SMT}. Leveraging the observation that front-end HTML5 pattern attributes and input types mirror the back-end validation of a web application, we illustrate how to integrate patterns like emails, zip codes, phone numbers, and maximum lengths into scanning and fuzzing. With our dynamic regex extraction, we can also handle regex-based JavaScript validation, used in popular libraries such as jQuery Validate. This increases our coverage of web applications, as we can bypass form validation while still generating inputs containing XSS injections, tokens for

taint tracking, or other side constraints required by the scanner. Our evaluation on 8 821 patterns extracted from popular websites demonstrates that Black Ostrich yields a 175% improvement in coverage and a 45% improvement in vulnerability detection compared to the average of the other scanners. We analyze the use of patterns in open-source web applications from GitHub. We perform a case study on three of the projects and showcase improved coverage specifically thanks to our string solving capabilities. Finally, we have used our implementation of the ECMA Regular Expression standard of JavaScript to analyze a condensed set of harvested email validation patterns. This served the dual purpose of illustrating the correctness of our implementation, as we were able to find matching strings for the vast majority of the analyzes regular expressions, as well as showing that 26% of email validation patterns would be vulnerable to XSS injections without further sanitization. We also illustrated a significant diversity among the commonly used patterns, suggesting that many website authors indeed hand-craft email validating patterns.

Bibliography

[1] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. Norn: An SMT solver for string constraints. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 462–469. Springer, 2015.

[2] R. Amadini. A survey on string constraint solving. *CoRR*, abs/2002.02376, 2020.

[3] Anonymous, authors of this paper. Data and code used in this study. Will be released upon publication. Submitted in HotCRP as additional material.

[4] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401. IEEE, 2008.

[5] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[6] C.-P. Bezemer, A. Mesbah, and A. van Deursen. Automated security testing of web widget interactions. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 81–90. ACM, 2009.

[7] J. Birget. State-complexity of finite-state devices, state compressibility and incompressibility. *Math. Syst. Theory*, 26(3):237–269, 1993.

[8] P. Bisht, T. L. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *CCS*, pages 607–618. ACM, 2010.

[9] A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[10] T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*, 3(POPL):49:1–49:30, 2019.

[11] Common Crawl Foundation. Common Crawl. `https://commoncrawl.org`.

[12] M. Contributors. Html: Hypertext markup language. entry `<input type="email">`, 2021.

[13] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[14] L. M. de Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[15] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *USENIX Security Symposium 12*, pages 523–538, 2012.

[16] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz. Ligre: Reverse-engineering of control and data flow models for black-box xss detection. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 252–261. IEEE, 2013.

[17] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 37–48, 2014.

[18] B. Eriksson, G. Pellegrino, and A. Sabelfeld. Black Widow: Blackbox Data-driven Web Scanning. In *S&P*, 2021.

[19] J. Fonseca, N. Seixas, M. Vieira, and H. Madeira. Analysis of field data on web security vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 11(2):89–100, 2014.

[20] B. D. A. G. and M. Stampar. sqlmap, 2021.

[21] V. Geffert, C. A. Kapoutsis, and M. Zakzok. Complement for two-way alternating automata. *Acta Informatica*, 58(5):463–495, 2021.

[22] V. Geffert and A. Okhotin. Transforming two-way alternating finite automata to one-way nondeterministic automata. In E. Csuhaj-Varjú, M. Dietzfelbinger, and Z. Ésik, editors, *MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proc., Part I*, volume 8634 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2014.

[23] GitHub. web-application · GitHub Topics. `https://github.com/topics/web-application`.

[24] W. G. Halfond, S. R. Choudhary, and A. Orso. Penetration testing with improved input vector identification. In *2009 International Conference on Software Testing Verification and Validation*, pages 346–355. IEEE, 2009.

[25] J. Harband and K. Smith. ECMAScript 2020 language specification, 11th edition, 2020. `https://262.ecma-international.org/11.0/`.

[26] L. Holík, P. Janku, A. W. Lin, P. Rümmer, and T. Vojnar. String constraints with concatenation and transducers solved efficiently. *Proc. ACM Program. Lang.*, 2(POPL):4:1–4:32, 2018.

[27] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition.* Pearson international edition. Addison-Wesley, 2007.

[28] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, pages 247–256, 2006.

[29] C. A. Kapoutsis. Removing bidirectionality from nondeterministic finite automata. In J. Jedrzejowicz and A. Szepietowski, editors, *MFCS 2005, Gdansk, Poland*, volume 3618 of *Lecture Notes in Computer Science*, pages 544–555. Springer, 2005.

[30] C. A. Kapoutsis and M. Zakzok. Alternation in two-way finite automata. *Theor. Comput. Sci.*, 870:75–102, 2021.

[31] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A Solver for String Constraints. In *ISSTA*. ACM, 2009.

[32] D. J. C. Klensin. Application Techniques for Checking and Transformation of Names. RFC 3696, Feb. 2004.

[33] R. E. Ladner, R. J. Lipton, and L. J. Stockmeyer. Alternating pushdown and stack automata. *SIAM Journal on Computing*, 13(1):135–155, 1984.

[34] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, 2019. List available at `https://tranco-list.eu/list/N5QW`.

[35] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV*, pages 646–662, 2014.

[36] B. Loring, D. Mitchell, and J. Kinder. Sound regular expression semantics for dynamic symbolic execution of JavaScript. In K. S. McKinley and K. Fisher, editors, *Proc. of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 425–438. ACM, 2019.

[37] A. Mesbah, E. Bozdag, and A. Van Deursen. Crawling ajax by inferring user interface state changes. In *2008 Eighth International Conference on Web Engineering*, pages 122–134. IEEE, 2008.

[38] OWASP. Owasp zed attack proxy (zap), 2020.

[39] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *International Symposium on Recent Advances in Intrusion Detection*, pages 295–316. Springer, 2015.

[40] T. S. Rocha and E. Souto. Etssdetector: A tool to automatically detect cross-site scripting vulnerabilities. In *2014 IEEE 13th International Symposium on Network Computing and Applications*, pages 306–309, Aug 2014.

[41] Sarosys LLC. Framework - arachni - web application security scanner framework, 2019.

[42] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *S&P*, 2010.

[43] J. C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM J. Res. Dev.*, 3(2):198–200, 1959.

[44] M. Trinh, D. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *CCS*, pages 1232–1243, 2014.

[45] W3C. Html 5.2, 2021. `https://www.w3.org/TR/2021/SPSD-html52-20210128/`.

[46] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a Z3-based string solver for web application analysis. In *ESEC/SIGSOFT FSE*, pages 114–124, 2013.

Appendix

C.I The Built-in Email Validation of HTML5

```
[a-zA-Z0-9.!#$%&'*+\/=?^_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])
    ?(?:\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*
```

Listing C.1: The MDN regex for validating email addresses. It should be equivalent to the validation implemented by browser vendors for the `email` input type. [12]

C.II Case Study: Finding Vulnerable Email Regexes

In this section we illustrate the viability of our implementation of ECMA regexes in a string solver by applying it to the analysis of regexes found in the wild.

Browsers implementing the HTML5 `input` type also implement syntactic email validation equivalent to the regex in Listing 1, henceforth referred to as MDN [12]. This validation under-approximates the permissive requirements of the IETF internet standards summarized in RFC3696, disallowing some technically valid email addresses [32]. This is no doubt based on a pragmatic assumption that most people do not have addresses like the (technically valid) `"<script>alert(1)</script>"@example.com`.

Users of the `email` input type can add additional validation using the `pattern` attribute if they have further requirements. It is also possible for a web designer to forego the built-in validation by not using the `email` input type, and supplying a `pattern` attribute with their validation logic.

This section investigates how real-world regexes used to validate email address inputs relate to the built-in validation of web browsers. We also investigate the security implications of sharing regexes for validation between the front-end and back-end of a web application without modification. This has implications on security, as the semantics of the `pattern` attribute are different from the ones of most regex engines.

In particular, we ask these three research questions: *(i)* How many validation regexes would accept an XSS attack string rejected by MDN? *(ii)* How many validation expressions impose stricter constraints than MDN, rejecting some string accepted by it? *(iii)* If the `pattern` validation regex is reused for validation in a back-end, how often would this let through an XSS attack string?

We investigate these questions on a collection of 825 unique email-validating regexes. We obtain this collection by de-duplicating the significantly larger set of 869 347 patterns collected in Section C.5.1. We further filter out patterns where the name or id attributes of the input element contained the string “email”. This means that the data set contains both validation patterns used *in addition* to MDN and patterns used *instead of* it. Finally, we remove 2 patterns that use anchors incorrectly, leaving us with a total of 825.

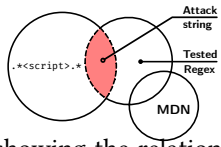


Figure C.7: A Venn diagram showing the relationship between the string sought in Section C.II.1, MDN, and the set of strings accepted by the pattern being experimented on. Note that no string containing a `<script>` tag is accepted by MDN! Sizes have no meaning.

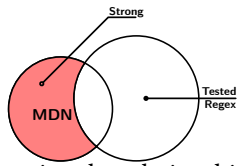


Figure C.8: A Venn diagram showing the relationship between the string sought in Section C.II.2 and MDN. Here we seek some string that is rejected by the pattern but accepted by MDN. Sizes have no meaning.

Table C.3 summarizes the results of all three investigations. Note that the acceptance of an email address containing a `<script>` tag is neither necessarily in violation of the IETF standards nor is it a guaranteed vulnerability in the application.

We conduct two experiments to illustrate that custom email validation regexes are simultaneously both weak and strong compared to the baseline MDN, in the sense that they simultaneously both allow an XSS attack string rejected by MDN and reject some input that is accepted by MDN. For an illustration of these concepts, please see Figure C.7 and Figure C.8 respectively.

C.II.1 Vulnerable Patterns

To find potential sources of vulnerabilities not permitted by MDN, we instruct Ostrich to find a string s for each regex ρ out of the 825 collected ones such that s matches ρ, contains a `<script>` tag, and does not match MDN. For an illustration, see the Venn diagram in Figure C.7. This illustrates that we can find at least one meaningfully different string that would pass the validation expression but not the built-in validation of web browsers. This has interesting implications under the assumption that a regex used for validation in the front-end of a web application would have a high probability of being reused for the corresponding validation in the back-end. The extended implications of this hypothesis are further investigated in Section C.II.3.

This experiment finds 215 potentially vulnerable regexes (satisfiable). 572 regexes are either fully subsumed by MDN, i.e., equivalent or strictly weaker, or will not allow a `<script>` tag. The remaining 38 regexes trigger syntax errors during parsing in Ostrich and had to be discarded from the study. All matching strings are validated against the `RegExp` class in NodeJS 15.14.0, and all but 2 are found to match, meaning

that our translation of the ECMA semantics as described in Section C.3 is precise for the most part and leaving us with a total of 213 vulnerable regexes.

C.II.2 Strong Patterns vs MDN

We also want to investigate if the patterns used are enforcing meaningfully stronger constraints on their inputs than MDN in the sense of further restricting the inputs of the form field. To do this, we invoke Ostrich on each regex ρ to find a string not matching ρ but matching MDN. An illustration can be seen in Figure C.8. This experiment yields a larger number of matching strings: 745, suggesting that these constraints are either typically used to narrow the set of allowed inputs or based on under-approximating expressions like `.+@.+\..+`. The occurrence of negative lookaheads to eliminate free email hosts further supports the first part of the hypothesis, suggesting that the author intended to block these, a typical semantic validation not captured by the built-in syntactic email validation. One of the generated strings contained the domain "me.com", from a regex meant to block email addresses from common free email hosts. All the other examples include exclamation points, ampersands, single quotes, pluses, or slashes, which are allowed before the `@`-sign by MDN but commonly disallowed by custom validation expressions whether by omission or on purpose. Note that these are examples found by Ostrich and that it does not imply that these patterns all represent the same constraints.

C.II.3 Vulnerabilities When Sharing Code

For HTML5 forms the semantics of the `pattern` attribute state that the regex must match the full input. This is not the case for most regex engines used in back-ends, where it is sufficient to only match a substring. If the same regex is used for validation both at the front-end and at the back-end this would mean that the validation in the back-end is potentially weaker than the one in the front-end. Specifically, this would be the case for regexes without anchors matching the beginning and end of the string.

To verify how common the use of such regexes is, we execute an experiment where we expand regexes not containing anchors (`^$`) with catch-all expressions (`.*`), in an opposite fashion to the logic of Section C.5.2. As the semantics of regexes are rather complicated, we expand them naively by simply replacing any expression beginning and ending with the expansion, if they did not contain either anchor, allowing post-solving validation to flag the edge cases where the regexes were more complicated. In other words, `.+@.+` would become `.*.+@.+.*`. We could find an attack for 531 of the modified regexes, and could verify actual vulnerability for 502 of them; an increase of 289 from the 213 vulnerable ones we found in Section C.II.1.

C.II.4 Summary

Email-validating HTML5 `pattern`s are diverse. It is common for them to be both weak compared to the built-in validation, and to refine the built-in validation with additional constraints, as can be seen in Table C.3. While the latter case is hardly a

	In `pattern`	In back-end
Accepts <script>	213	502
Rejects MDN-valid input	745	n/a

Table C.3: Comparison of crawled email validation patterns to the built-in HTML5 validation.

cause for concern for the security of the application, the use of redundant validation expressions is also suggestive of code reuse. If that would be the case, differences in semantics between the HTML `pattern` attribute and all common regex engines would make the validation at the back-end weaker than the one at the front-end. This implies that security vulnerabilities will be present in many web applications if the strings are reused unsanitized.

Finally, these experiments illustrate that our encoding of the ECMA regex semantics is both versatile and performant enough to solve both substring matching and (non-) intersection for real-world regexes, many of them highly complex and all of them harvested from real websites. Only 38 (unable to parse) plus two (semantically invalid) out of the 825 regexes are untranslatable into our encoding.

C.III Details of Section C.3

We define the semantics of augmented regular expressions recursively, in the style of semantics for temporal logic:

Definition C.5 (Augmented regex semantics)**.** Let $w = w_0 w_1 \dots w_n$ be a string in Σ^* of length $\ell(w) = n + 1$, and let $w(i, j)$ be the segment of the string starting at i and ending at j (excluded). If $j > \ell(w)$, then $w(i, j) = w(i, \ell(w))$ and for every $j \leq i$ we have $w(i, j) = \varepsilon$, i.e., the empty string. Moreover, we denote with $w(i)$ the i-th symbol of w, which is undefined if $i \geq \ell(w)$. We say that w is accepted by $\rho \in \mathcal{R}$, and write $w \in L(\rho)$ iff $w, (0, \ell(w)) \models \rho$ holds, which is inductively defined on the structure of ρ as follows:

- $w, (i, j) \models \emptyset$;
- $w, (i, j) \models \varepsilon$ iff $i = j$;
- $w, (i, j) \models \sigma$ iff $j = i + 1$ and $j \leq \ell(w)$ and $w(i) = \sigma$;
- $w, (i, j) \models \rho^*$ iff $i = j$ or there exist indexes $k_0, \dots, k_m$ such that $i = k_0 \leq k_1 \leq \dots \leq k_m = j$ and $w, (k_l, k_{l+1}) \models \rho$ for each $l \in \{0, \dots, m - 1\}$;
- $w, (i, j) \models \rho_1 \cdot \rho_2$ iff there exists $i \leq k \leq j$ such that $w, (i, k) \models \rho_1$ and $w, (k, j) \models \rho_2$;
- $w, (i, j) \models \rho_1 + \rho_2$ iff $w, (i, j) \models \rho_1$ or $w, (i, j) \models \rho_2$;
- $w, (i, j) \models (> \rho)$ iff $i = j$ and there exists $i \leq k \leq \ell(w)$ such that $w, (i, k) \models \rho$;
- $w, (i, j) \models (\overline{>} \rho)$ iff $i = j$ and for all $i \leq k \leq \ell(w)$ we have $w, (i, k) \not\models \rho$;

- $w,(i,j) \models (<\rho)$ iff $i = j$ and there exists $0 \leq k \leq i$ such that $w,(k,i) \models \rho$;
- $w,(i,j) \models (\overline{<}\rho)$ iff $i = j$ and for all $0 \leq k \leq i$ we have $w,(k,i) \not\models \rho$;
- $w,(i,j) \models \hat{}$ iff $i = j = 0$;
- $w,(i,j) \models \$$ iff $i = j = \ell(w)$.

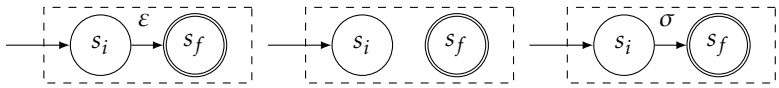


Figure C.9: Schematic representation of automaton construction recursive steps for the atomic cases: ε, $\emptyset$ and $\sigma \in \Sigma$.

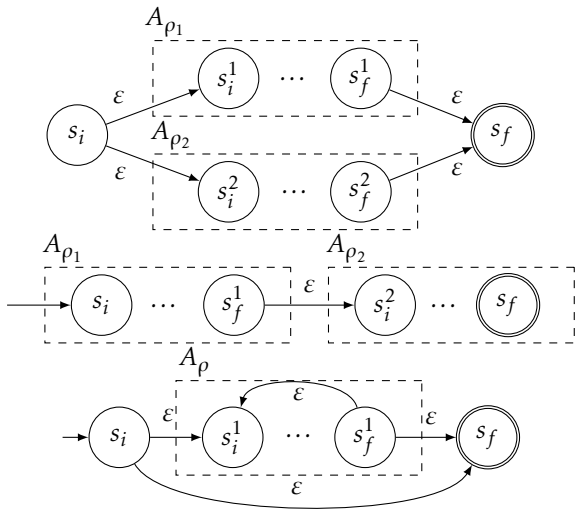


Figure C.10: Schematic representation of automaton construction recursive steps for for: $\rho_1 + \rho_2$, $\rho_1 \cdot \rho_2$ and ρ^*.

1: **Input**: A regex $\rho \in \mathcal{R}$
2: **Output**: a regex $\rho' \in \mathcal{R}$
3: **procedure** TRFM(ρ)
4: **case** ρ **of**
5: $a \mid \varepsilon \mid \emptyset$: **return** ρ
6: ρ^*: **return** TRFM(ρ)*
7: $\rho_1 \cdot \rho_2$: **return** TRFM(ρ_1)$\cdot$TRFM(ρ_2)
8: $\rho_1 + \rho_2$: **return** TRFM(ρ_1) + TRFM(ρ_2)
9: $(>\rho)$: **return** $(>$TRFM(ρ)$)$
10: $(\overline{>}\rho)$: **return** $(\overline{>}$TRFM(ρ)$)$
11: $(<\rho)$: **return** $(<$REV(ρ)$)$

```
    (≲ρ):  return (≲ REV(ρ))
    ^:  return (≲ Σ+)
    $:  return (≳ Σ+)
    end case
end procedure
```

We now describe in detail the procedure for generating the 2AFA_{SMT}.

Given $\rho \in \mathcal{R}$, we first transform the input in "normal form" (in linear time in the size of ρ) by calling TRFM(ρ): this intuitively replaces anchors ^ and \$ with $\lesssim \Sigma$ and $\equiv \gtrsim \Sigma$ respectively and it reverses subexpressions inside look-behinds.

Then, the automaton is build from the bottom up by calling AUT on TRFM(ρ), which works recursively on the structure of the formula. The subroutine ATOMICAUT returns an automaton as in Figure C.9, while STARAUT, ALTAUT, CONCAUT or LOOKAUT return automata as in the Figure C.10 and C.2 where s_i^j and s_f^j are the initial and final state of the automaton returned by the recursive call on ρ_j. The routine NEG takes an automaton and returns its complement in polynomial time, e.g., by following the technique in [21].

```
Input: A regex ρ ∈ R
Output: 2AFA_SMT A_ρ
procedure AUT(◊, ρ)
    case ρ of
    a | ε | ∅:  return ATOMICAUT(◊, ρ)        ▷ Transitions are δ^< if ◊ is <, δ^>
otherwise
    ρ*:  return STARAUT(AUT(◊, ρ))
    ρ1·ρ2:  return CONCAUT(AUT(◊, ρ1), AUT(◊, ρ2))
    ρ1 + ρ2:  return ALTAUT(AUT(◊, ρ1), AUT(◊, ρ2))
    (>ρ):  return LOOKAUT(AUT(>, ρ))
    (≳ρ):  return LOOKAUT(NEG(AUT(>, ρ)))
    (<ρ):  return LOOKAUT(AUT(<, ρ))
    (≲ρ):  return LOOKAUT(NEG(AUT(<, ρ)))
    end case
end procedure
```

Proof. [Proof of Theorem 1] The proof is modular and the main part requires to prove the following statement: for each w, i, j and ρ, $w,(i,j) \models \rho$ iff AUT($>,\rho$) has an accepting run on w starting from position i in w and AUT($<$, REV(ρ)) has an accepting run on w starting from position j. From the semantics of regexes, $w,(i,j) \models (<\rho)$ iff $i = j$ and there exists $0 \leq k \leq i$ such that $w,(k,i) \models \rho$. The automaton LOOKAUT(AUT($<$, REV(ρ))) is as in Figure C.2 where A_ρ is the (backward) automaton for ρ. Since the first transition is a $\varepsilon_\forall$, the run expands into two pairs $\{(s_i^1, i), (s_{f,2}, i)\}$. The pair $(s_{f,2}, i)$ accepts iff $i = j$, namely there is nothing left to read to the right: indeed, by definition of 2AFA_{SMT}, any symbol σ read from $s_{f,2}$ leads to a non-accepting sink state (which we do not explicitly show in the figures for the sake of readability). Let us now consider the upper path: by inductive hypothesis, A_ρ (without the Σ-loop on $s_{f,1}$) accepts iff $w,(k,i) \models \rho$ for some $0 \leq k \leq i$. This means that the run will eventually reach $s_{f,1}$ before, or at, index 0. Since $s_{f,1}$ is now

$$\frac{}{\emptyset \xrightarrow{\emptyset} \emptyset} \qquad \frac{}{\epsilon \xrightarrow{\emptyset} \epsilon} \qquad \frac{}{\sigma \xrightarrow{\emptyset} \sigma} \qquad \frac{\rho \xrightarrow{\emptyset} \mathsf{r}}{\rho^* \xrightarrow{\emptyset} \mathsf{r}^*}$$

$$\frac{\rho \xrightarrow{F} \mathsf{r} \qquad \rho' \xrightarrow{F'} \mathsf{r}' \qquad r \notin F, l \notin F'}{\rho \cdot \rho' \xrightarrow{F \cup F'} \mathsf{r} \cdot \mathsf{r}'} \qquad \frac{\rho \xrightarrow{F} \mathsf{r} \qquad \rho' \xrightarrow{F'} \mathsf{r}'}{\rho + \rho' \xrightarrow{F \cup F'} \mathsf{r} + \mathsf{r}'}$$

$$\frac{\rho \cdot \Sigma^* \xrightarrow{F} \mathsf{r} \qquad \rho' \xrightarrow{F'} \mathsf{r}'}{(>\rho) \cdot \rho' \xrightarrow{F \cup F' \cup \{r\}} \mathsf{r} \cap \mathsf{r}'} \qquad \frac{\rho \cdot \Sigma^* \xrightarrow{\emptyset} \mathsf{r} \qquad \rho' \xrightarrow{F} \mathsf{r}'}{(\overline{>}\rho) \cdot \rho' \xrightarrow{F \cup \{r\}} \overline{\mathsf{r}} \cap \mathsf{r}'}$$

$$\frac{\rho \xrightarrow{F} \mathsf{r} \qquad \Sigma^* \cdot \rho' \xrightarrow{F'} \mathsf{r}'}{\rho \cdot (<\rho') \xrightarrow{F \cup F' \cup \{l\}} \mathsf{r} \cap \mathsf{r}'} \qquad \frac{\rho \xrightarrow{F} \mathsf{r} \qquad \Sigma^* \cdot \rho' \xrightarrow{\emptyset} \mathsf{r}'}{\rho \cdot (\overline{<}\rho') \xrightarrow{F \cup \{l\}} \overline{\mathsf{r}} \cap \mathsf{r}'}$$

$$\frac{\rho \xrightarrow{F} \mathsf{r}}{\rho \cdot \text{^} \xrightarrow{F \cup \{l\}} r \cap \epsilon} \qquad \frac{\rho \xrightarrow{F} \mathsf{r}}{\$ \cdot \rho \xrightarrow{F \cup \{r\}} r \cap \epsilon}$$

Table C.4: Rules translating ECMA regexes to textbook regexes

a sink accepting state thanks to the Σ-loop, the run from $(s_{i,1}, i)$ accepts iff there exists $0 \leq k \leq i$ such that $w, (k, i) \models \rho$. An analogous argument holds for $\textsc{aut}(>, \rho)$.

■

From 2AFA$_{SMT}$ to S-2AFA$_{SMT}$. We transform the former into the latter by performing the following steps. First, we multiply the states: for each s of the 2AFA$_{SMT}$, which in general has $>$ and $<$ both incoming and outgoing transitions, we have four states of the S-2AFA$_{SMT}$ for each pairwise combinations of those ($S^>$ and $s_>$ in S-2AFA$_{SMT}$ are generated from the initial and sink states of the 2AFA$_{SMT}$). Such states are connected with $\varepsilon_\exists$ transitions and therefore the original semantics is preserved. In the second step we remove the ε transitions by exploiting bidirectionality. Indeed, we can simulate a $\varepsilon_\exists$ transition from, e.g., state s to s' by: adding a special state s''; having $>$ transitions from s to s'' for every symbol $\sigma \in \Sigma$ and then having $<$ transitions from s'' to s' again for every symbol. This easily generalizes to set of states and $\varepsilon_\forall$ transitions as well, but special care is needed at the end of the word where $>$ are not allowed. To handle those cases, we actually introduce special markers at the beginning and end of the words, which we do not explicitly include here for the sake of readability. Lastly, the S-2AFA$_{SMT}$ only accepts at the end of the word, as it only has $F^>$ state, and this can be obtained by moving to an $F^>$ accepting state from former accepting states $F^<$ by means of $\delta^>_\exists$ transitions.

C.IV Partial Translation from ECMAScript Regexes to Textbook Regexes

This section introduces our partial, semantics-preserving translation of ECMAScript regexes SMT-LIB (or textbook) regexes (Section C.3.5). Our translation is defined in terms of translation statements $\rho \xrightarrow{F} \mathsf{r}$, which express that the ECMA regex ρ is equivalent to the textbook regex r, modulo side-effects F. The set F marks cases in which the effect of ρ can extend beyond the beginning or end of a string due to look-arounds or anchors.

Definition C.6. A translation statement $\rho \xrightarrow{F} \mathsf{r}$ is *valid* if:

1. $L(\rho) = L(\mathsf{r})$;
2. $F \subseteq \{l, r\}$;
3. if $l \notin F$, then for all $w \in \Sigma^*$ it holds that $L(w\rho) = L(w\mathsf{r})$;
4. if $r \notin F$, then for all $w \in \Sigma^*$ it holds that $L(\rho w) = L(\mathsf{r}w)$.

Because not all ECMA regexes can be translated straightforwardly to textbook regexes, we introduce a set of rules for deriving translation statements (Table C.4). A rule allows the derivation of the statement in the conclusion (below the bar) assuming some set of premises (above the bar). As a convention, we assume that all rules are applied modulo the associativity of concatenation. For the complete translation of an expression, a derivation tree can be constructed in which each leaf is closed through an axiom, viz., a rule with no premises.

The first three rules in the table are axioms and they describe the translation of atoms $\emptyset, \epsilon, \sigma$. The next three rules capture the Kleene star, concatenation, and alternation, and recursively translate the sub-expressions first. The rule for concatenation has the side condition that the left-hand side ρ must not contain look-aheads or anchors whose effect might extend to the right ($r \notin F$), and effects of the right-hand side ρ' must not extend to the left ($l \notin F'$). Absent those side conditions, compositional translation of $\rho \cdot \rho'$ is not possible.

The last three lines in Table C.4 describe the translation of look-arounds and anchors. For look-ahead $(>\rho)\cdot\rho'$, the sub-expressions can be translated separately, and the results conjoined, and similarly for the other versions of look-around. Anchors imply the absence of leading or trailing characters, and can also be translated to intersection.

Example 3. We illustrate the rules by translating the expression $(>\Sigma^* a)(\gtrdot \Sigma^* b)\Sigma^*$, describing strings that contain at least one a but no b. The resulting regex is the intersection $(\Sigma^* a \Sigma^*) \cap \overline{\Sigma^* b \Sigma^*} \cap \Sigma^*$.

$$\dfrac{\dfrac{\vdots}{\Sigma^* a \Sigma^* \xrightarrow{\emptyset} \Sigma^* a \Sigma^*} \quad \dfrac{\dfrac{\vdots}{\Sigma^* b \Sigma^* \xrightarrow{\emptyset} \Sigma^* b \Sigma^*} \quad \dfrac{\overline{\Sigma \xrightarrow{\emptyset} \Sigma}}{\Sigma^* \xrightarrow{\emptyset} \Sigma^*}}{(\gtrdot \Sigma^* b)\Sigma^* \xrightarrow{r} \overline{\Sigma^* b \Sigma^*} \cap \Sigma^*}}{(>\Sigma^* a)(\gtrdot \Sigma^* b)\Sigma^* \xrightarrow{r} (\Sigma^* a \Sigma^*) \cap \overline{\Sigma^* b \Sigma^*} \cap \Sigma^*}$$

Lemma C.2 (Soundness of translation)**.** The root of a closed derivation tree using rules from Table C.4 is a valid statement.

Proof. To prove soundness of the rules in Table C.4, we need to show that each of the rules in the table preserves the validity of translation statements: if the statements above the bar are valid, then the statement below the bar is valid. This is straightforward for most of the rules.

As an example, we consider the rule for concatenation,

$$\frac{\rho \xrightarrow{F} \mathsf{r} \qquad \rho' \xrightarrow{F'} \mathsf{r}' \qquad r \notin F, l \notin F'}{\rho{\cdot}\rho' \xrightarrow{F \cup F'} \mathsf{r}{\cdot}\mathsf{r}'}$$

Assume that all statements in the premise are valid, and $r \notin F, l \notin F'$. There are two directions to be shown for $L(\rho{\cdot}\rho') = L(\mathsf{r}{\cdot}\mathsf{r}')$:

"$\subseteq$": Assume $w \in L(\rho{\cdot}\rho')$, which by definition means that $w(0,k) \in L(\rho)$ and $w(k,\ell(w)) \in L(\rho')$ for some k with $0 \leq k \leq \ell(w)$. Write $u = w(0,k)$ and $v = w(k,\ell(w))$. We need to show that $u \in L(\mathsf{r})$ and $v \in L(\mathsf{r}')$. For this, observe that also $w = uv \in L(\rho v)$ and $w = uv \in L(u\rho')$. Because $r \notin F, l \notin F'$, we have $L(\rho v) = L(\mathsf{r}v)$ and $L(u\rho) = L(u\mathsf{r}')$, and therefore $u \in L(\mathsf{r})$ and $v \in L(\mathsf{r}')$. This finally implies that $w = uv \in L(\mathsf{r}{\cdot}\mathsf{r}')$.

"$\supseteq$": Assume $w \in L(\mathsf{r}{\cdot}\mathsf{r}')$, which directly implies that w can be split into $w = uv$ such that $u \in L(\mathsf{r})$ and $v \in L(\mathsf{r}')$. Choose k such that $u = w(0,k)$ and $v = w(k,\ell(w))$. By assumption, we again have $uv \in L(\mathsf{r}v) = L(\rho v)$ and $uv \in L(u\mathsf{r}') = L(u\rho')$, and therefore also $(uv)(0,k) \in L(\rho)$ and $(uv)(k,\ell(w)) \in L(\rho')$. In combination, this implies $w \in L(\rho{\cdot}\rho')$. ∎

Approximation. In practice, we apply the translation rules in Table C.4 greedily but might have to resort to over-approximation when regexes are encountered that cannot be translated precisely. Problematic sub-expressions of a regex can in such cases be rewritten by removing look-arounds or anchors with function $\mathit{rem} : \mathcal{R} \to \mathsf{R}$ recursively defined as follows:

$$\begin{array}{lcl}
\mathit{rem}(\rho) & = & \rho \text{ for } \rho \in \{\emptyset, \varepsilon, \sigma\} \\
\mathit{rem}(\rho^*) & = & \mathit{rem}(\rho)^* \\
\mathit{rem}(\rho_1 \,\square\, \rho_2) & = & \mathit{rem}(\rho_1) \,\square\, \mathit{rem}(\rho) \text{ for } \square \in \{\cdot, +\} \\
\mathit{rem}(\triangledown\rho) & = & \varepsilon \text{ for } \triangledown \in \{>, \not>, <, \not<\} \\
\mathit{rem}(\rho) & = & \varepsilon \text{ for } \rho \in \{\hat{}, \$\}
\end{array}$$

We remark that $\mathit{rem}(\rho)$ semantically over-approximates ρ:

Lemma C.3. Let $\rho \in \mathcal{R}$, then $L(\rho) \subseteq L(\mathit{rem}(\rho))$.

Proof. By induction on the structure of ρ. Case of *atom* is trivial. Case of ρ^*: by IH, $L(\rho) \subseteq L(\mathit{rem}(\rho))$, therefore $(L(\rho))^* = L(\rho^*) \subseteq (L(\mathit{rem}(\rho)))^* = L(\mathit{rem}(\rho)^*)$. Case of $\rho = \rho_1 + \rho_2$: by IH, $L(\rho_i) \subseteq L(\mathit{rem}(\rho_i))$ for $i \in \{1,2\}$. It follows that $L(\rho_1 + \rho_2) = L(\rho_1) \cup L(\rho_2) \subseteq L(\mathit{rem}(\rho_1 + \rho_2)) = L(\mathit{rem}(\rho_1)) \cup L(\mathit{rem}(\rho_2))$. Case of $\rho = \rho_1 \cdot \rho_2$ analogous to the previous case with $\cdot$ instead of $\cup$.

For the case of $\triangledown\rho$, we notice from the semantics that since $i = j$, $L(\triangledown\rho)$ is either $\emptyset$ or $\{\varepsilon\}$, and in both cases the hypothesis holds. Concerning the anchors, we have

$L(\hat{}) = L(\$) = L(rem(\hat{})) = L(rem(\$)) = L(\varepsilon) = \{\varepsilon\}$. ■ To avoid incorrect solutions due to approximation, we verify that constructed strings are indeed accepted by the original ECMA regex.

C.V Example Input for Ostrich (Section C.5.3.2)

The following SMT-LIB script will ask an SMT solver to construct a string `w` of length at most 30 that satisfies the email regular expression from the introduction, while also including the payload string `<script>alert(1)</script>`:

```
(declare-const w String)
(assert (str.in-re w (re.from_ecma2020
          '.*@.*\.[a-z]{2,3}')))
(assert (str.contains w
          "<script>alert(1)</script>"))
(assert (<= (str.len w) 30))
(check-sat)
(get-model)
```

C.VI Testbed Code

We show our design of the testbed's client-side code in Section C.VI and the server-side in Section C.VI.

```
<html>
<body>
<form action="PATTERN_HASH.php" method="GET">
<input  pattern="PATTERN" type="MODE_TYPE" name="MODE_NAME"
    >
<br>
<input type="submit" name="submit">
</form>
```

Listing C.2: Client-side code for each pattern. PATTERN_HASH is replace with the hash of the pattern and PATTERN is replaced by the actual pattern.

```
  // Check type pattern for email and URL
  // ...
  // Check pattern
  if( preg_match("/^(?:PATTERN$)$/u", SCANNER_INPUT) ) {
    // Reflect for XSS
    echo $_GET['text_input_field'];
    // log successful solve
  } else {
    // log failed solve
```

```
    echo "Failed to match";
  }
```

Listing C.3: Server-side code. PATTERN is replaced by the actual pattern.

C.VII Client-side form validation

Table C.5 contains a list of validation attributes in HTML5.

General attributes	
`type`	The type of the input, most relevant are: `color`, `date`, `datetime-local`, `email`, `month`, `number`, `password`, `range`, `tel`, `text`, `time`, `url`, `week`.
`required`	Field is non-empty
Attributes for string inputs	
`minlength`	Minimum length of input string
`maxlength`	Maximum length of input string
`pattern`	A regular expression defining the expected textual input.
Attributes for numerical inputs	
`min`	Minimum value of a numerical input, which can be of integer or fractional/decimal type
`max`	Maximum value of a numerical input
`step`	Step size of a numerical input: the input value has to be a multiple of the given (integer or fractional) number

Table C.5: HTML5 client-side form validation constraints

C.VIII ExpoSE JavaScript Template

We use the following code to create JavaScript files for each pattern. The placeholder `[[PATTERN]]` is replaced with the actual pattern. Running ExpoSE will tell us if `Reachable_regex_solved` is reachable and thus if the pattern is solvable.

```
var S$ = require('S$');
var a = S$.symbol('A', '');
var re = new RegExp("^(?:" + [[PATTERN]] + ")$" );
if (a.match(re)) {
  throw 'Reachable_regex_solved';
}
```

C.IX Algorithm for validation-aware scanning

This algorithm shows the crawling phase of the scanner. For the attack phase line 6 is repeated for each payload the scanner uses. We assume a *node* to be a general object that can be scanned, like a URL or JavaScript event. For each input element, we check for a validation constraint, like a pattern attribute, and try to solve it. For payloads, we can add the payload as the second argument to `solveContraint`, instructing it to solve the pattern and include the payload string.

```
Data: Target url
nodes = scanPage(url); while node = nodes.pop() do
    if node.type == FORM then
        for element in node.elements do
            if validationConstraint(element) then
                element.value = solveConstraint(element, "");
            end
        end
    end
    nodes += scanNode(node);
end
```

Figure C.11: Crawling algorithm with pattern solver.

Browser Extensions

D

Hardening the Security Analysis of Browser Extensions

Abstract. Browser extensions boost the browsing experience by a range of features from automatic translation and grammar correction to password management, ad blocking, and remote desktops. Yet the power of extensions poses significant privacy and security challenges because extensions can be malicious and/or vulnerable. We observe that there are gaps in the previous work on analyzing the security of browser extensions and present a systematic study of attack entry points in the browser extension ecosystem. Our study reveals novel password stealing, traffic stealing, and inter-extension attacks. Based on a combination of static and dynamic analysis we show how to discover extension attacks, both known and novel ones, and study their prevalence in the wild. We show that 1 349 extensions are vulnerable to inter-extension attacks leading to XSS. Our empirical study uncovers a remarkable cluster of "New Tab" extensions where 4 410 extensions perform traffic stealing attacks. We suggest several avenues for the countermeasures against the uncovered attacks, ranging from refining the permission model to mitigating the attacks by declarations in manifest files.

D.1 Introduction

Modern web browsers allow users to customize and improve their browsing experience by installing browser extensions. The functionalities of these extensions can range from modifying the aesthetics of websites to blocking advertisements, adding accessibility features, or security and privacy features. Using these functionalities *malicious extensions* routinely steal information from unknowing users [16, 20] and thrive on fake content injection like fake ads [21].

Similar to mobile apps, the extensions are mainly installed from app stores, such as the Chrome Web Store. Google is continuously removing malicious extensions from the Web Store [21]. Yet, new malicious extensions continue emerging [16, 32]. Although extensions submitted to the Chrome Web Store are subject to analysis and vetting, the problem with automatically analyzing extensions is that detecting different threats requires different methods.

Table D.1: Summary of the attacks versus the ecosystem presented in this paper.

Attack	Subattack	Attacker	Victim	In wild	Section
Password	Chrome autofill	Extension	Extension/User/Web page	Novel	Section D.5.1.1
	Virtual keyboard	Extension	Extension/User/Web page	Novel	Section D.5.1.2
Traffic		Extension	User	4 410	Section D.5.2.1
Inter-extension	Collusion	Extension	User	Benign	Section D.5.2.2
	History poisoning	Extension/Web page	Extension	1 349	Section D.5.3.1
	Code execution	Extension/Web page	Extension	1 349	Section D.5.3.2
	Fingerprint	Extension	User	10 785	Section D.5.3.3

Threat Model. The power of extensions poses privacy and security challenges. Extensions can both read sensitive information directly [14, 18] and indirectly by redirecting network traffic. For example, a malicious extension can use JavaScript to read passwords from the DOM or listen to network traffic. There are further underexplored classes of attacks where malicious extensions can also attack other extensions, for example, to steal their internal data, like todo-notes or stored passwords.

The challenge is not only to find malicious extensions but also *vulnerable extensions*. For example, an attacker could trick an extension with access to the user's cookies to send the cookies to the attacker. Previous work has uncovered several classes of attacks and vulnerabilities related to browser extensions (discussed in detail in Section D.9). For example, Kapravelos et al. [22] find malicious extensions trying to steal data or modify security headers. Somé [37] discovers code execution vulnerabilities in extensions through static and manual analysis. In this scenario, a malicious website attacks a vulnerable extension. Attacking the implementation in the browser is also possible. For example, Buyukkayhan et al. [7] exploit the lack of isolation mechanisms that Firefox used to implement in its browser extension ecosystem.

Yet there are gaps in the previous work when it comes to analyzing the security of the entire browser extension ecosystem. Our paper is a step towards filling the gaps in the security analysis of extensions. We accomplish this by performing a systematic study of the extension ecosystem, including extensions' assets, attackers, and possible interaction methods. The benefit of our approach is the wider threat model.

Approach. We propose a systematic approach to hardening the security analysis of browser extensions. The main thrust of our systematization is a systematic study of attack entry points in the browser extension ecosystem. This leads us to both discovering novel attacks and analyzing known ones from a wider attacker model perspective. We group all the attacks by the actors involved to define the attacker, victim, attack surface, and target asset. Based on the attack we use a combination of static and dynamic analysis to detect insecure extensions and in some cases synthesize payloads. We download all the 133,365 extensions from the Chrome Web Store and test our detection mechanism on the extensions. We search for characteristics of our novel attacks on the web and confirm their novelty by finding no evidence of attackers using them in the wild so far.

Attacks. We present novel vectors that extensions can use to attack both the user and other extensions. We divide them into three categories: password stealing, traffic stealing, and inter-extension attacks. We summarize these attacks in Table F.1.

Password stealing: We develop a new method for actively stealing passwords, circumventing Chrome's protection. Chrome tries to protect against password stealing by not adding the password to the page DOM before a user interacts with the page. To circumvent this, an extension can change the type of the password field to text and capture a screenshot.

Traffic stealing: By analyzing extensions from the Web Store, we find extensions that are actively stealing search queries by redirecting traffic.

Inter-extension attacks: We create novel methods for detecting potentially vulnerable extensions that can be attacked by other extensions. We detect this by analyzing how inter-extension message passing and poisoning shared resources can lead to Cross-Site Scripting (XSS). In addition, we show that inter-extension message passing can also fingerprint extensions.

Empirical study. Our findings show that 4 410 extensions, totaling over 120 000 downloads, are actively stealing search queries from users. We also detect 1 349 extensions being vulnerable to takeover using XSS from malicious extensions. Furthermore, this group of extensions is also vulnerable to history poisoning attacks leading to XSS while 2 829 are vulnerable to HTML code injection via history poisoning. In the latter case, Content Security Policy (CSP) protects them from XSS. Our new method for fingerprinting extensions also improves the state-of-the-art by adding 162 new extensions.

Countermeasures. We propose countermeasures for each class of the aforementioned attacks. In brief, for the password attacks, we consider that extensions that want to take screenshots should declare a concrete permission (for example the screenshot API `captureVisibleTab`) similar to the need for the `desktopCapture` permission. For traffic stealing, Chrome is already in the process of adopting a new version of the manifest file where extensions will have to declare in advance how they will handle users' requests [15]. Finally, for the inter-extension attacks, we propose to either make mandatory the definition of the `externally_connectable` key in the manifest file or if not, change the security-by-default option, and if such a key is not in the manifest, then the extension will not handle any external message.

Contributions. The paper offers the following contributions:

1. We present a systematic study of attack entry points in the browser extension ecosystem in Section D.3.
2. We describe our methodology based on a combination of static and dynamic analysis to discover attacks in Section D.4.
3. We study the prevalence of attacks in the wild and two novel attacks: password stealers and inter-extension history poisoning in Section D.5.
4. We perform a detailed case study of the popular "New Tab" extensions in Section D.6.

5. We present countermeasures to the identified problems in Section D.7.

We release our implementation and example extensions[1].

D.2 Background

While implementation details differ between the main browsers (e.g., Firefox and Chromium-based browsers), the overall architecture for browser extensions is similar.

Chrome isolates the execution of browser extensions in different environments for security reasons [4]. Due to this, *message passing* is used. We can classify message passing, depending on who the sender is, into 1) scripts provided by the web page; 2) content scripts of the extensions, and; 3) background pages.

If the sender is a web page, it can use the `postMessage` method to send a message to any other script also running in the web page context. Also, web pages can send direct messages to background pages of extensions by using one-time requests (`sendMessage(<ext_id>)` from `runtime` and `tabs` APIs). Browser extensions will handle these messages by implementing `onMessageExternal`.

These event listeners are triggered if and only if the browser extensions define the `externally_connectable` key in the manifest file and explicitly add the web page in the `matches` sub-key. Since extensions cannot establish this external communication by themselves, the only way they have to reply to external messages is in the body of the event listener by using `sendResponse` or `postMessage`, depending on whether the communication is one-time or long-lived respectively. Using the same procedure, Chrome allows cross-extension messaging.

If the sender is the content script of an extension, it can send messages to scripts provided by web pages (or to other content scripts) by using `runtime.postMessage` method. It can also send direct messages to the background of the same extension using one-time requests. The background of the extension will handle messages coming from the content scripts of the same extension by implementing `onMessage()`. Also, content scripts can send external messages to other extensions using the method explained before, i.e., as if they were scripts coming from web pages.

If the sender is the background, it can use `postMessage()` to send messages to the content scripts of the same extension and `runtime.sendMessage(<ext_id>)` for cross-extension messaging.

[1] `https://www.cse.chalmers.se/research/group/security/hardening-extensions/`

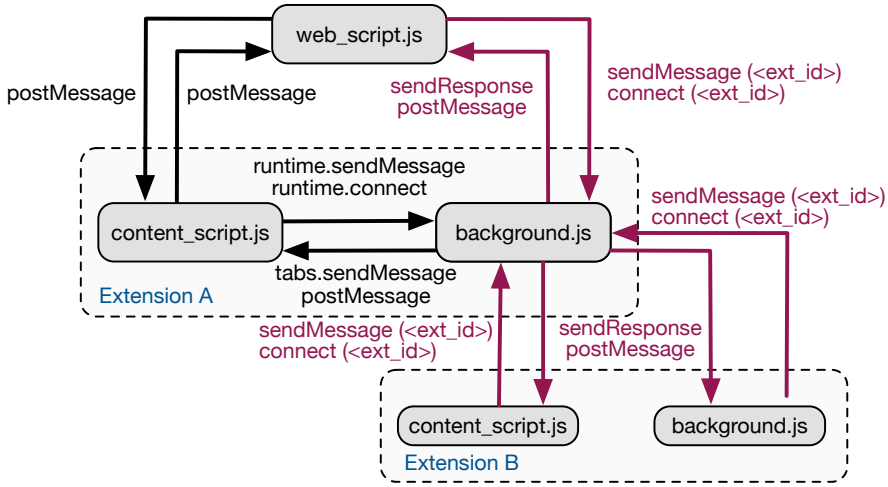


Figure D.1: Message passing in browser extensions.

In Figure D.1, we include a summary of the message passing in browser extensions. In the figure, we represent two extensions (Extension A and B) and simplify the methods to send and receive information between isolated worlds (content scripts, background pages, and web scripts) as explained before.

D.3 Threat Model

We define a **vulnerable extension** as an extension that can lose control of any of its confidential or integrity-sensitive assets to another actor. For example, another malicious extension can send a message resulting in code execution. On the contrary, we say that an extension is **malicious** when it controls assets from another actor. For example, a malicious extension could steal the user's password, which is an asset. We define both the actors and the assets in the description of the entire ecosystem in Section D.4.1.

There are two main **entry points**, which we explore in the following sections, that attackers can use to exploit browser extensions: 1) shared resources, and; 2) message passing.

D.3.1 Shared Resources

We consider shared resources to be all the elements that extensions have access to, e.g., DOM content, history, Web Accessible Resources (WARs), and bookmarks. The DOM can be used by scripts and extensions to communicate and share data [13].

Table D.2: Each row shows which communications are possible based on the IDs and matches defined in the manifest.

#	∃	externally_connectable		Communication	
		IDs	matches	Web pages	Extensions
1	X	–	–	X	∀
2	✓	$[ID_1,\ldots,ID_n]$	$[URL_1,\ldots,URL_n]$	$[URL_1,\ldots,URL_n]$	$[ID_1,\ldots,ID_n]$
3	✓	$[ID_1,\ldots,ID_n]$	X	X	$[ID_1,\ldots,ID_n]$
4	✓	X	$[URL_1,\ldots,URL_n]$	$[URL_1,\ldots,URL_n]$	X
5	✓	X	X	X	X
6	✓	["*"]	$[URL_1,\ldots,URL_n]$	$[URL_1,\ldots,URL_n]$	∀
7	✓	["*"]	X	X	∀

D.3.2 Message Passing

As explained in Section D.2, in message passing, we can distinguish between messages coming from scripts of the same extension and external ones. For messages coming from external scripts, we propose a novel and methodological way to extract from the manifest of the extensions how vulnerable they are.

External message passing. To know to what extent the extensions are vulnerable due to message passing, we use the optional `externally_connectable` key. Such a key can include two optional keys, `ids`, and `matches`. The former indicates the list of extension IDs whose messages are handled, whereas the latter for web pages instead.

If the `externally_connectable` key is not defined in the manifest (1^{st} row in Table D.2), all extensions can send messages but webpages cannot. On the contrary, if the key is defined there can be 6 possible options depending on the values of the two lists. We include in Table D.2 all the possible cases of (not) defining the `externally_connectable` key in the manifest. Note that the most secure option is when the key is defined but none of the lists are provided (5^{th} row in Table D.2). By analyzing the manifest file of all the extensions, we found that 6,417 extensions define `externally_connectable` key whereas 126,948 do not, meaning there are more than 100 000 extensions that accept external messages coming from other extensions.

Content scripts message passing. Even though message passing using content scripts is an attack vector, as recently demonstrated [37], we consider that this is a particular case of the DOM shared resource entry point. Extensions can react to the event fired by either web pages or by other extensions when they send a message.

D.4 Methodology

In this section, we explain in detail our method for discovering malicious as well as potentially vulnerable extensions.

D.4.1 Identifying entry points

To harden the analysis we first perform a systematic analysis of the extension ecosystem, including assets, attackers, and interaction methods. We use a top-down approach and start by analyzing the actors, followed by the different assets they possess, the attack surface, and finally the entry points into the extension.

Ecosystem. The browser extension ecosystem allows for rich interaction between three classes of actors: *users*, *web pages*, and *extensions*.

Security Assets. We define security assets as important assets related to sensitive information and program control flow. The main asset of the user in our model is sensitive information, such as history, cookies, passwords, and the user's online activity. Passwords are also custody of web pages, making them an important asset to web pages too. Finally, extensions' assets include both confidential user-generated data, like todo-notes and passwords, and control flow (e.g., code execution). For example, an extension stealing sensitive data like users' history is malicious as it is reading a confidential asset (history).

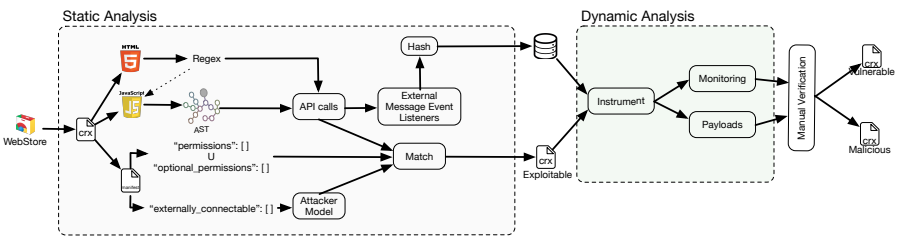


Figure D.2: Static and dynamic analysis combination to determine possible attacks and detect malicious and potentially vulnerable extensions.

Attack surface. We model the attack surface by close inspection of the interaction methods available to the actors. For shared resources, we consider DOM content, history, bookmarks, WARs, and cookies. These could potentially be used to carry malicious payloads. In Figure D.1 we show the possible message passing interactions between extensions and web pages that make up the second part of the attack surface for the two actors. Note that both web pages and extensions can use message passing but in the inter-extension case the messages can go directly to the high-privileged background scripts making such vulnerabilities a larger threat compared to attacks from web pages. We also further refine the attack surface dynamically by inspecting the manifest of the extension. By comparing the manifests of extensions to Table D.2 we can efficiently filter out attack vectors.

Entry points. The final step is to extract all the entry points from the attack surface and analyze them. We use a combination of static and dynamic analysis based on the type of entry points to do this.

D.4.2 Combining Static and Dynamic Analysis

We crawled the Web Store in Jan 2020 and downloaded all the 133 365 browser extensions. To evaluate our approach on as many extensions as possible we include both old and young extensions. As it can be seen in Figure D.2, to detect both malicious and potentially vulnerable extensions, we split our methodology into i) static, and; ii) dynamic analysis. Finally, we installed the potentially vulnerable extensions and confirmed whether they are exploitable or not.

To improve efficiency and harden the analysis we combine both static and dynamic analysis. By combining them we no longer need to dynamically test all extensions, which would require more resources. In many cases, we can do this without losing precision, e.g., by statically analyzing the manifest we know if an extension can manipulate web requests to perform traffic stealing. If the extension lacks the permission we can skip the dynamic analysis.

Static Analysis. During the static analysis, we analyze `externally_connectable` according to Table D.2. We also parse both permissions and the optional permissions of the extensions to know what sensitive information the extension has access to. This takes about 12 hours on a normal workstation.

We use Esprima[2], a powerful library to perform lexical (tokenization) and syntactic (parsing) analysis of JavaScript, to generate the Abstract Syntax Tree (AST) of the JavaScript files and extract the external message event listener's code. For HTML files, we use regular expressions instead to extract the code.

We find 53 177 potentially malicious extensions containing either the API `runtime.connect()` or `runtime.sendMessage()` functions. We also find 11 595 extensions that are susceptible to being abused. That is, they implement event listeners for external message passing functions, e.g., `onMessageExternal()`.

After, we compute the SHA-256 hash of the event listener functions and group extensions by their hashes, obtaining 570 different implementations. The use of cryptographic hashes, as opposed to fuzzy hashes like ssdeep [24], ensures that the functions are the same. We discuss the implications of this in Section E.6. Then, we chose one extension from each of the 570 groups and dynamically analyzed them. Note that since each extension in the group has the same function it does not matter which one we pick.

Dynamic Analysis. Based on the static analysis we extracted extensions that can be further tested for dynamic analysis. The dynamic analysis consists of three major steps: 1) instrumentation; 2) monitoring, and; 3) payload generation. The exact implementation of these differ from attack to attack and is explained in more detail in Section D.5.

In general, the extensions are statically instrumented to log the line numbers being executed. This is done by inserting fetch instructions on each line, like the following: `fetch("http://localhost/inst/'+token()+'/line/'+(line_number)+'")`. For each attack, we also hooked API calls, including the arguments, and exfiltrate them by using similar instructions and encodings.

[2] `https://esprima.org`

To monitor we use a web server that listens to these fetch instructions. Based on the requests we can determine the trace of the extension's execution and if any security and privacy-relevant APIs were executed.

Finally, the last part is generating payloads or instructions to bring the extension to an interesting state. We used puppeteer [31], which is a tool for controlling Chrome, to load the extension we wanted to test and any other extensions we developed to interact with it. From here the instructions depend on the type of attack. For example, our extension can send inter-extension messages to the other extension. We can load pre-configured chrome profiles to test history poisoning attacks or navigate the browser to test traffic stealing. Dynamically testing one extension takes around 1 minute, but many can be done in parallel and after the static filtering we are only dynamically executing a subset of all extensions.

Manual Verification. When the static or dynamic analysis marks an extension as either potentially vulnerable or malicious we make sure to manually verify it. To manually verify a malicious extension we run the original version of the extension, i.e., without instrumentation code and other modifications, in a normal Chrome browser, not via puppeteer. From there we verify that the same malicious behavior, for example, traffic stealing, is still present. Similarly, for vulnerable extensions, we also use the original extension and load it in Chrome together with a suitable attacker extension we create.

To avoid manually testing every vulnerable or malicious extension, we cluster them using *DeDup.js* [30]. *DeDup.js* allows us to find all extensions that have the same malicious or vulnerable JavaScript files. We also check the manifests to ensure the JavaScript files are loaded and executed. We discuss the risk of possible false positives from this approach in the discussion in Section E.6.

D.5 Discovering Attacks and Vulnerabilities

In this section, we present the novel attacks that we designed as well as both the vulnerable and the malicious extensions we found in the wild. Our novel attacks are presented in Section D.5.1, the attacks used by malicious extensions are presented in Section D.5.2, while the vulnerable extensions are in Section D.5.3. We analyze these attacks and vulnerabilities in Chrome but discuss how they apply to other browsers in Section D.8.4.

D.5.1 Novel Attacks by Malicious Extensions

In this section, we detail two novel attacks we designed to steal users' credentials. As these attacks are designed by us and not discussed in prior works, we did not expect to find any of these in the wild. We did not find any extensions using these attacks, further supporting their novelty.

D.5.1.1 Password stealer

A password-stealing extension can attack any actor in the attacker model, i.e., other extensions, web pages, or users. We have seen previous attacks [42] where extensions inspect the DOM to steal passwords.

In this paper, we present a novel active password-stealing approach where the extension actively visits different domains to extract the passwords. A novel component to this is a bypass of Chrome's protection against autofill-scraping [43]. Note that this protection is so far only in Chromium-based browsers, i.e., not in Firefox or Safari yet. The attack still works in Firefox but not in Safari as it requires the user to click a pop-up before the password is added to the field. Chrome protects the user from autofill attacks by waiting for user interaction before adding the password to the DOM. However, we can edit the underlying password element to bypass the protection. Our attack changes the type of the password element to `text` instead of `password`. Although the value is still not added to the DOM, the asterisks are converted into text. Our extension takes a screenshot of the page to exfiltrate the image with the password. Note that taking screenshots of the pages only requires the `<all_urls>` permission, which is used by over 18 000 extensions, including popular extensions like Grammarly with over 10 million downloads. Furthermore, no special screenshot permissions or user interaction are required.

Making the attack stealthy to the user is an orthogonal problem. We use a pop-under attack which creates a new window under the active one. In this new window, we load a page and take a screenshot. This approach makes the attack stealthy on both macOS and Ubuntu. Another approach can be to modify the style of the DOM to make it harder for the user to see the unmasked password. In addition, the attack takes less than five seconds to steal a password in our tests, with possible variation due to network speed, making it hard to stop even if the user notices it.

D.5.1.2 Virtual keyboard attack

To increase the security of the user's credentials, some online services include virtual keyboards on the screen for the users to avoid directly writing the passwords and thus protect them from keylogger attacks. For this attack to work, the attacker has to implement an click event listener using `document.onClick`. Later on, in an offline analysis, the attacker matches both the coordinates and the screenshot to get not only the clicked elements but also the order in which they were clicked. This login system is popular among banks, e.g., ING bank in Spain, France, Australia, BNP Paribas, and La Banque Postale in France, without the option for the users to use an alternative.

D.5.2 Malicious Extensions in the Wild

In this section, we present extensions attacking users or other extensions.

D.5.2.1 Traffic stealing

Extensions have the power to cancel, redirect, modify request and response headers, and supply authentication. This allows them to implement features like ad-blocking by matching URLs against deny-lists. The permissions needed to modify ongoing network requests are `webRequest` and `host_permissions`, and `webRequestBlocking`. Using these permissions and APIs, an extension can attack the user by intercepting general search queries, for example to Google or Yahoo, redirect these back to their server, and finally have their server redirect the user back to a search engine. Without the user noticing, their search traffic is stolen.

Results. To detect malicious extensions exploiting this attack, we analyzed those that interact with network requests. In particular, extensions that used the `webRequestBlocking` permission and defined a listener for `onBeforeRequest` were dynamically analyzed. We clustered extensions based on the SHA-512 hash of the file responsible for the traffic stealing.

Before dynamically executing the extension, we instrumented it to send the list of filtered URLs to our control server at runtime. These are the potential URLs the extension can interact with. We then compared this to a list of domains, plus query parameters, we wanted to test against. The three domains (`www.google.com`, `www.yahoo.com`, and `www.bing.com`) were picked based on a short pilot study of malicious extensions we found, we discuss improvements to this method in Section E.6. For each match, we dynamically executed and analyzed the extension. To detect network requests we use `page.setRequestInterception` in Puppeteer. Using this we notice if the extension introduces new requests or redirects.

Example. We found 4 410 extensions abusing this method to steal traffic from users. We discuss these in detail in Section D.6.

D.5.2.2 Collusion

Extensions can communicate directly to other extensions by sending messages. The possibility for message passing depends on the `externally_connectable` definition (see Table D.2). Malicious extensions can abuse message passing to share permissions, allowing multiple low permission extensions to combine their permissions. This would allow them to stay under the radar.

Results. To detect collusion we looked for message passing between extensions. As can be seen in Figure D.1, the `sendMessage` function takes an extension ID as a parameter. Therefore, we scanned each extension for extension IDs in their code.

Extension IDs follow a simple pattern, 32 lower case characters. We first scanned each file using this regular expression in python, `re.findall("[a-z]{32}")`. This returns all the strings that could potentially be other extensions. Next, we compared all the potential extension IDs with the real ones in the dataset we are using. To further filter the list we removed the extensions that do not use message passing APIs. This helps to remove extensions that simply link to other extensions, without directly messaging them. Based on this, we created our "collusion map" which we then manually analyzed to understand why the extensions interact with each other.

Example. In this paper, we did not find any colluding extensions that could be classified as malicious. However, we did find two extensions colluding with each other to share permissions. The first one[3] defines three permissions while the second one[4] requires no permissions. The second one then asks the first one for data.

D.5.3 Vulnerable Extensions in the Wild

In this section, we present the vulnerable extensions we found during our analysis.

D.5.3.1 History poisoning

History poisoning is a novel attack we present which targets extensions working with users' browser histories. Malicious extensions, or web pages, can poison the browser history with code injection payloads. The vulnerable extensions usually present the user with an overview of their history. This can range from exact history to most visited or recently closed tabs. By adding HTML code to the title of a web page, a web attacker can gain content script code execution if the history titles are not sanitized correctly. The exploitability of this attack depends on the type of history poisoning necessary. Changing the most visited pages is difficult for a web attacker to accomplish. However, adding a single entry to recent history or closed tabs only requires a redirection.

Results. To detect potentially vulnerable extensions due to this attack, we first extract extensions with the `history` permission. Once a vulnerability is found we automatically mark other extensions which share the same file and use the history permission. Later, in the dynamic phase, we loaded the history with visits to pages with HTML and JavaScript payloads in the title. These payloads are explained in Section D.6. We then loaded the extension and if a payload is executed we conclude that the extension is vulnerable. Based on the CSP in the manifest, this can either imply full XSS or be limited to HTML injection.

Example. In Section D.6 we present a detailed analysis of extensions vulnerable to this attack.

D.5.3.2 Code execution

If an extension uses dynamic code execution functions, such as `eval`, without safety precautions it can potentially be abused by other web pages and extensions.

The impact of this attack depends on the context of the code execution. The highest impact context is the background scripts where, if exploited, an attacker would gain access to all the APIs the extension has permission to use. The impact in the DOM context depends on the connection to the background. By executing code in the DOM context, the attacker gains access to all the APIs available for the content scripts. For instance, the attacker can make use of the `sendMessage` function and impersonate the legitimate content scripts of the extensions, allowing the attacker

[3] dnclbikcihnpjohihfcmmldgkjnebgnj (version 1.3.6)

[4] bepofoammpdjhfdibmlghoaljkemineg (version 0.1.4)

to send messages to the background as if she were the content scripts, affecting the confidentiality, integrity, and availability of the extension.

Results. To detect code execution vulnerabilities, we extracted the code from the message passing functions in Figure D.1. We did this by analyzing and extracting listeners like `onMessageExternal`. From this, we clustered the extensions based on the SHA-512 hash of the code of these functions. For each cluster, we extracted the API calls from the listener functions to check if they either execute code dynamically, e.g., by `eval`, or interact with shared resources, e.g., local storage. We then checked these API calls against the permissions that the extension uses. Finally, we analyzed the clusters with API calls to determine if they can lead to code execution.

To find more extensions we search for others containing the same file, and if the permissions are correct, we mark them as vulnerable. This could lead to false positives if another extension has the same file but does not use it. However, in our manual testing, we found no such cases.

Example. In this paper, a group of 1 349 extensions was vulnerable to code execution attacks. We discuss them in-depth in Section D.6

D.5.3.3 Fingerprinting

Knowing which extensions a user has installed can allow both extension and web page attackers to degrade the privacy of a user. Multiple methods of browser extension fingerprinting have been proposed. Sjösten *et al.* [36] used WARs; Starov *et al.* [38] identified extensions by the changes they perform to the DOM; [23] *et al.* do so by sending messages and waiting for their response; recently, [25] *et al.* used CSS styles. However, extensions with the "management" permission can use the `chrome.management.getAll` function to get a list of all installed extensions. We show how extensions can bypass the "management" permission and get this list. Note that malicious extensions do not require any extra capabilities or permissions to bypass the permission and perform the attack.

Our technique is based on the `externally_connectable` property. Recall this allows extensions and web pages to send messages to another extension using inter-extension messaging. If that property is not in the manifest, by default the target extension will accept all messages coming from extensions. However, if the property is defined, then two main properties can be defined: `matches` and `ids`. Both are allowed-lists where only the web pages and extensions defined can send direct messages to the target extension.

Results. To detect how many extensions are fingerprintable due to external messages, we parsed the manifest file of all the 133 365 we downloaded as of January 2020. 126,948 extensions **do not** define the `externally_connectable` key in the manifest. Remember that if this key is not in the manifest file but the extensions implement listeners for external communications, e.g., `onMessageExternal`, web pages cannot send direct messages through message passing functions but other extensions can.

From the 126k extensions, we statically check how many of them implement any of the methods to handle external connections and got a lower bound of 11 595 potentially fingerprintable extensions. Later, we automatically installed them all—

Table D.3: Browser extensions fingerprinting.

	Methodology	Browser Extensions	
		Fingerprintable	Non-Fingerprintable
2019	This paper	10 785 (8.0%)	665
	WARs[36]	10 919 (8.1%)	531
	WARs [36] ∩ Bloat [38]	10 977 (8.2%)	473
	This paper ∩ [36] ∩ [38]	11 147 (8.3%)	303

146 could not be analyzed because the manifest had syntactic errors and the bowser could not install them—and certified that 10 785 use the `sendResponse()` function to send a response back to the sender.

We then coded two fingerprinting attacks proposed by 1) Sjösten *et al.* [36], where extensions are fingerprintable due to the WARs they publicly expose, and; 2) Starov *et al.* [38], where extensions are fingerprintable due to bloat (changes they automatically perform over the DOM). In Table D.3 we summarized our findings and the combination with prior work in the field. Despite being aware of a recently published work [23], we could not reproduce the proposed attack being therefore impossible for us to corroborate their findings and include them in the final result. After executing the three fingerprinting attacks (third row of Table D.3), we successfully fingerprinted 11 147 extensions (approximately 8.3% of the total extensions) where 303 were not possible to do so with these techniques. The reason for not being possible to be fingerprinted is because either they do not send a message back using the `sendResponse()` method, they do not define WARs, or they do not automatically modify the web content provided by the server.

D.6 New Tabs Case Study

Our empirical study has uncovered a remarkable cluster of "New Tab" extensions. The main characteristic of these extensions is that they override the new tab functionality in the browser. When the user opens a new tab in the browser, this is replaced by the one the new tab extension created. Such new tabs usually have a search bar with some arbitrary wallpaper backgrounds. Some extensions also add widgets like todo-notes, weather reports, email, a list of the most visited sites, and bookmarks management among others.

Our analysis has flagged many vulnerable and malicious extensions in this category, justifying further investigation. In the following section, we analyze them in more detail.

Traffic stealing. We found 4 410 extensions, with a combined 176 thousand downloads, that steal search queries from users while posing as new tab extensions. There is a common file in all these extensions called `search-overwrite.js` which implements an event listener that is fired just before a web request is sent (`chrome.`

Table D.4: Distribution of file versions for `search-overwrite.js` and domains used to steal queries.

URL	# Extensions	SHA[0:3]
s.tablovel.com	1740	026
www.explorenewtab.com	1178	1f8
www.newtabprobe.com	560	191
www.newtabexplore.com	539	513
www.lovelychrometab.com	255	381
www.newtabexplore.com	82	6d9
www.themefornewtab.com	40	c49
www.newtabwallpapers.com	8	7e1
www.newtabwallpapers.com	7	5dc
www.newtabprobe.com	1	732

`webRequest.onBeforeRequest.addListener`). In such a function, all the extensions block the ongoing request and redirect it to different URLs, like `s.tablovel.com`. We realized that, in addition to stealing queries from popular search engines like Google, a subset of them, containing 2 588 extensions, also steal queries from other new tab extensions, e.g., from `redirect.lovelytab.com`.

Regarding the `search-overwrite.js` file, from all the 4 410 extensions, we computed all the SHA-512 hashes of such a file in different extensions and found that there are 10 unique versions. In Table D.4 we show the distribution of the files and the URL used to steal the queries. As can be seen, the two most popular versions of that file are used by a majority of the extensions.

Code execution. In addition to finding malicious new tab extensions, we also found multiple vulnerable ones. In particular, we found that 1 349 extensions were vulnerable to XSS attacks from other extensions. At the time of download, these extensions had over 73 million downloads combined, making them quite popular. However, we do believe many of these downloads were fraudulent and some extensions have lost many of their downloads since then. This attack depends on three factors: 1) the extensions allow external messages; 2) the external messages are stored and reflected in the extension, and; 3) the CSP in the manifest allows for JavaScript execution. The problem in these extensions is that data from the `localStorage` is reflected unsanitized, as is the case for the "Peppa Pig HQ Wallpapers New Tab"[5] with over 3 000 downloads. For the 1 349 extensions, all of the above criteria were met. We also found two other groups of new tab extensions, totaling 2 829 extensions, which met all but the last criterion. This means that XSS is no longer possible but HTML injections still are, allowing attackers to inject ads or meta redirects, effectively taking over the new tab extension.

[5] cikheolhmcgdkmblgmfkgkcgffllddaem (version: 3.2) now removed.

History poisoning. A popular function with new tab extensions is showing both recently and most visited websites. We set up a local server and installed the extensions in the browser. The browser is running a profile with multiple poisoned shared resources, i.e., before running the extension, history data are added to the profile. We found that if we change the `<title>` tag of a page in the history, we were able to remotely execute code in the user's browser.

As an example, we use "Halloween Backgrounds New Tab"[6] and the title payload `<script>alert(1)</script>`. When the extension reads our web page from the most visited list of the user, which happens when the user opens a new tab of the browser or when the user launches the browser, the payload is executed. We found that the same group of extensions being vulnerable to inter-extension XSS is also vulnerable to XSS from history poisoning. In total, all 1 349 are vulnerable to XSS and 2 829 to HTML injection from history poisoning.

Evasion. We discovered that New Tab extensions actively practice evasion mechanisms, making their detection challenging for purely dynamic mechanisms. It turned out that 53 of them share a file that implements a time bomb logic that waits five days after installation before starting to redirect traffic.

D.7 Countermeasures

In this section, we briefly discuss possible countermeasures for the attacks we discovered.

Password stealing. Stopping malicious extensions from stealing passwords is complicated. Many legitimate password manager extensions need the capability to both read and write passwords to the DOM. Thus, simply blocking access to password fields is not viable. To stop the more nefarious type of password-stealing, which is active password-stealing where no user input is needed, Chrome decided to hide autofilled passwords from the DOM until the user interacts with the page, which our attack bypasses with screenshots.

To protect against extensions screenshotting passwords there are three avenues. First, warning the users about the screenshot being taken, similarly to what Chrome does for full desktop screenshots already. A similar approach for screenshots using `chrome.tabs.captureVisibleTab`, would clearly show the password being leaked to the user.

The second approach is for the browser to hide the password before taking the screenshot. As the browser knows which field it inserts the saved password into it can either obfuscate or fully remove this field before taking the screenshot. Then after the capture is completed, add back the field again. The third approach is to create a new permission such that extensions that want to use the `captureVisibleTab` function should declare in advance such permissions so the user has to specifically approve such privileges.

[6] jckojlfhehjnlkdoiojpmkjjoojbgfjl (version: 0.1.8.4) now removed.

Traffic stealing. With the adoption of the new manifest version 3, the permission `webRequestBlocking` will be moved to the `chrome.declarativeNetRequest` API [15]. If extensions would want to block web requests by using the `chrome.webRequest` API, they will have to declare that in advance in a set of rules. With this, traffic stealing will not be solved but extensions will have to explicitly declare the purpose in advance, being easier to detect and block by either Google or users.

Inter-extension attacks. In this paper, we conclude that one of the most effective methods to avoid undesired extensions to send messages to others is using the `externally_connectable` key in the manifest, similar to what Somé [37] suggested. We developed a script that automatically inserts `externally_connectable` into the manifest of those extensions without it. After, we run the inter-extension attacks presented in this paper and certified that the extensions were not exploitable anymore due to external messages.

A common problem in both the extension being vulnerable to inter-extension XSS and history poisoning attacks was that they did not sanitize user-controlled input. These problems could have been avoided by properly sanitizing content before presenting it to the user, whether it is todo-notes or an overview of the browser history. There are many good JavaScript libraries available to sanitize input before adding it to the DOM, e.g., Both jQuery and DOMPurify.

For XSS specifically, strict CSP policies are very effective. Chrome already applies a default strict CSP policy protecting against inline XSS attacks. However, as we found in this paper, many extensions override this with weaker policies. Making it clearer to both the developers and users that weak CSP policies can lead to the extension and user data being compromised might help raise awareness of this problem.

D.8 Discussion

In this section, we discuss the limitations of our work and how we try to minimize both false positives and false negatives. We also discuss deviating results in the Safari browser.

D.8.1 Static analysis

There are some inherent limitations in using static code analysis. In the case of JavaScript files, there are two main aspects to consider: dynamic function execution and code obfuscation. Dynamically executed code inside an `eval()` statement would not be parsed and analyzed correctly. Obfuscated JavaScript code is another well-known problem. During our manual analysis, we did find a heavily obfuscated traffic stealing extension that used a base64-encoded string to evade static detection. Thus, even if we use basic regular expression searches in the code, they can be evaded. To improve this future static analysis approaches should focus more on deobfuscation and attempt to decode and decrypt data.

Due to dynamic code execution and obfuscation, our analysis may miss potentially malicious extensions resulting in false negatives. However, for the code in the extension that can be parsed, we ensure that the permissions and APIs we extract are in the extension's code.

D.8.2 Dynamic analysis

Before we dynamically executed the extensions, we instrumented them to relay useful information about API calls. However, the instrumentation is done statically, which inherits similar limitations as mentioned in the static analysis. In the case of traffic stealing detection, this means that we would not be able to acquire the exact filter list the extensions use. We handled this by dynamically testing the extension on all the URLs in our target list, which means that we tested everything but at the cost of worse performance.

For traffic stealing, we tested extensions against these three URLs that had been targeted by traffic stealers, `www.google.com`, `www.yahoo.com`, and `www.bing.com`. Similarly, by executing the extension multiple times on the same URL we could reduce the noise in terms of requests being made, thus lowering false positives. For example, visiting `www.google.com` might result in different requests between two visits. By making multiple visits both with and without the extension we could determine which requests are introduced by the extensions. Future work includes testing more URLs than the three we picked. This can lower the false negatives by potentially finding more malicious extensions.

D.8.3 Manual Analysis

To further minimize any false positives from the static and dynamic analysis we test the original extension in a plain Chrome browser. We test one extension from each cluster of extensions, where the cluster is all the extensions that have the same vulnerable or malicious JavaScript file. We also check that the manifests for all the extensions in the cluster contain the same file in the manifest to ensure it is not simply unreachable code.

D.8.4 Cross-browser

For each attack and vulnerability we recreated them in both Firefox and Safari. In general, they worked similarly in Firefox but failed in Safari. This is because browser extensions in Safari follow a slightly different model than both Chrome and Firefox. In Safari, extensions are composed of three main parts [2]: 1) a macOS app; 2) the browser extension—similar to Chrome and Firefox, and; 3) a native app extension that mediates between both the macOS app and the browser extension. In addition to that, the number of permissions available for extensions is limited in comparison to the other browsers. As a consequence, most of the attacks explained in Section D.5 are not exploitable in Safari but the password stealer.

D.9 Related Work

From the privacy point of view, we differentiate among approaches that focus on how browser extensions handle the (sensitive) information that they have access to [1, 10, 11, 12, 17, 41], approaches that demonstrate how extensions can be used to fingerprint users [23, 36, 38, 39, 40], and approaches that analyze the permissions of the extensions [4, 6, 7, 8, 9, 19, 28]. From the security point of view, extensions can be used to execute malicious code [3, 5, 37] and how extensions can cooperate to execute collusion attacks [33].

Often, these works pursue attacker models and capabilities based on concrete attacks, e.g., fingerprinting and information leaking. To complement the depth, our approach offers the breadth of systematically analyzing the attack surface for the browser extension ecosystem, identifying gaps that lead to discovering new attacks.

Mystique [12] proposes a powerful approach to taint privacy information in browser extensions by modifying the V8 engine of Chromium. This approach is suitable for detecting traffic stealing attacks, assuming the complexity of V8 does not break the soundness of taint tracking [44]. In this paper, we used a complementary technique based on a combination of static and dynamic analysis with clustering of similar extensions. While Mystique is powerful enough to find the traffic stealers, the benefit of our approach is its performance efficiency and independence of the browser engine.

We review the literature on the paper's three main themes: password-stealing, traffic stealing, and inter-extension attacks individually.

Password stealing. The security of password managers has been widely evaluated, resulting in discoveries of security flaws in the autofill functionality, being vulnerable to web pages managed by an attacker [26, 27, 35]. In this paper, we demonstrated how a malicious browser extension can transform the password field of the DOM into text, take a screenshot of the password, and transform it back without the user noticing.

Traffic stealing. In 2014, Hulk [22] was one of the first papers pointing out that extensions are sniffing users' communication traffic. The authors analyzed 48 332 extensions and marked 130 as malicious and 4 712 as suspicious. Seven years later, we realized that this technique is still being used by 4 410 extensions that capture users' searches and forward that information to external servers apart from performing the original request. To our surprise, most of these extensions fall into the "HD Wallpapers New Tab" group where they claim to modify Chrome's new tab page.

Inter-extension. Many authors focused on fingerprinting browser extensions. The common adversarial model is a web page that attempts to track users using the extensions they have running in the browser. Probing for WARs [36], detecting the changes the extensions automatically perform over the DOM [38], using message passing to know which extension replied to the message [40] or executing timing attacks [34] are a few examples of how browser extensions can be fingerprinted. In

our paper, apart from considering web pages, we also include browser extensions as attackers and knowing which extensions the user has without declaring the "management" permission.

Regarding collusion, very little effort has been made in this area in browser extensions. As far as we know, Saini *et al.* [33] were the first authors who demonstrated that legitimate extensions can collude to achieve malicious goals and showed how a malicious extension can misconfigure the browser or other extensions using message passing. Similarly, Buyukkayhan *et al.* [7] showed how the API that Firefox exposes to extensions, known as Cross Platform Component Object Model (XPCOM), can be exploitable by other extensions because of a lack of isolation mechanisms, being only able to be exploited in previous versions of Firefox. In contrast, even though Chrome isolates the execution of the extensions in different environments, we demonstrate that browser extensions in Chrome not only can modify others' configurations but also cooperate to achieve a common goal.

Pantelaios *et al.* [29] investigate 922,684 extensions and find 143 malicious extensions where 64 were still online. From these 143 extensions, they detected that 16 changed the search engine of the user. To do so, they use the users' feedback (a combination of rating and comments) and a clustering algorithm to classify and detect malicious extensions. With our systematic study of attack entry points and a combination of both static and dynamic analysis, we discover 4 410 that steal search queries of the user by redirecting them to external servers.

D.10 Conclusions

As a step toward filling the gap in the security analysis of the browser extension ecosystem, we have presented a systematic study of attack entry points leading us to novel methods for password stealing, traffic stealing, and inter-extension attacks. Because extensions are highly privileged, it pays off for the attacker to target vulnerable extensions, leading to possibilities of exfiltrating secrets and performing unauthorized modification. Combining static and dynamic analysis we have shown how to discover extension attacks and study their prevalence in the wild. Our findings indicate that 1 349 extensions are vulnerable to cross-extension messaging passing attacks leading to XSS. We also discovered a remarkable cluster of "New Tab" extensions where 4 410 extensions in this class perform traffic-stealing attacks. We have suggested countermeasures for the uncovered attacks.

Coordinated disclosure. Our disclosure report to Google includes not only malicious and vulnerable extensions but also recommendations on mitigating inter-extension attacks, as well as our findings on password and traffic stealing, which require browser support for the countermeasures. All of the 4 410 reported traffic stealers have now been deleted. We are in contact with Google about the rest, including the password stealer which is currently being triaged by their security team.

Acknowledgments. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice

Wallenberg Foundation, the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Facebook.

Bibliography

[1] A. Aggarwal, B. Viswanath, L. Zhang, S. Kumar, A. Shah, and P. Kumaraguru. I spy with my little eye: Analysis and detection of spying browser extensions. In *Euro S&P*, pages 47–61, 2018.

[2] Apple. Messaging between the app and javascript in a safari web extension. `https://developer.apple.com/documentation/safariservices/safari_web_extensions/messaging_between_the_app_and_javascript_in_a_safari_web_extension`, 2021.

[3] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vetting browser extensions for security vulnerabilities with VEX. *Commun. ACM*, 54(9), 2011.

[4] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*, 2010.

[5] A. Barua, M. Zulkernine, and K. Weldemariam. Protecting web browser extensions from javascript injection attacks. In *ICECCS*, pages 188–197, 2013.

[6] L. Bauer, S. Cai, L. Jia, T. Passaro, and Y. Tian. Analyzing the dangers posed by chrome extensions. In *CNS,*, pages 184–192, 2014.

[7] A. S. Buyukkayhan, K. Onarlioglu, W. K. Robertson, and E. Kirda. Crossfire: An analysis of firefox extension-reuse vulnerabilities. In *NDSS*, 2016.

[8] S. Calzavara, M. Bugliesi, S. Crafa, and E. Steffinlongo. Fine-grained detection of privilege escalation attacks on browser extensions. In *PLAS*, 2015.

[9] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the google chrome extension security architecture. In *USENIX Sec*, pages 97–111, 2012.

[10] W. Chang and S. Chen. Defeat information leakage from browser extensions via data obfuscation. In *ICICS*, pages 33–48, 2013.

[11] W. Chang and S. Chen. Extensionguard: Towards runtime browser extension information leakage detection. In *CNS*, pages 154–162, 2016.

[12] Q. Chen and A. Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *CCS*, page 1687–1700, 2018.

[13] Chrome. Content scripts. `https://developer.chrome.com/docs/extensions/mv2/content_scripts/`, 2019.

[14] Chrome. Chrome extensions permission model. `https://developer.chrome.com/extensions/declare_permissions`, 2020.

[15] G. Chrome. Migrating to Manifest V3. `https://developer.chrome.com/extensions/migrating_to_manifest_v3`, 2020.

[16] Google Pulls 49 Cryptocurrency Wallet Browser Extensions Found Stealing Private Keys, Apr. 2020. https://news.bitcoin.com/google-cryptocurrency-wallet-browser/.

[17] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *ACSAC*, pages 382–391, 2009.

[18] Firefox. Firefox extensions permission model. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/permissions, 2020.

[19] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *S&P*, pages 115–130, 2011.

[20] S. Jadali. DataSpii: The catastrophic data leak via browser extensions. https://securitywithsam.com/2019/07/dataspii-leak-via-browser-extensions/, 2019.

[21] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and lessons from three years fighting malicious extensions. In *USENIX Sec*, pages 579–593, 2015.

[22] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Sec*, pages 641–654, 2014.

[23] S. Karami, P. Ilia, K. Solomos, and J. Polakis. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *NDSS*, 2020.

[24] J. Kornblum. ssdeep - fuzzy hashing program. https://ssdeep-project.github.io/ssdeep/, 2021.

[25] P. Laperdrix, O. Starov, Q. Chen, A. Kapravelos, and N. Nikiforakis. Fingerprinting in style: Detecting browser extensions via injected style sheets. In *USENIX Sec*, 2021.

[26] Z. Li, W. He, D. Akhawe, and D. Song. The emperor's new password manager: Security analysis of web-based password managers. In *USENIX Sec*, 2014.

[27] X. Lin, P. Ilia, and J. Polakis. Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill. In *CCS*, 2020.

[28] L. Liu, X. Zhang, G. Yan, S. Chen, et al. Chrome extensions: Threat analysis and countermeasures. In *NDSS*, 2012.

[29] N. Pantelaios, N. Nikiforakis, and A. Kapravelos. You've changed: Detecting malicious browser extensions through their update deltas. In *CCS*, page 477–491, 2020.

[30] P. Picazo-Sanchez, M. Algehed, and A. Sabelfeld. Dedup.js: Discovering malicious and vulnerable extensions by detecting duplication. In *International Conference on Information Systems Security and Privacy (ICISSP)*, 2022.

[31] puppeteer. puppeteer. `https://github.com/puppeteer/puppeteer`, 2021.

[32] Reuters. Exclusive: Massive spying on users of Google's Chrome shows new security weakness. `https://www.reuters.com/article/us-alphabet-google-chrome-exclusive/exclusive-massive-spying-on-users-of-googles-chrome-shows-new-security-weakness-idUSKBN23P0JO`, 2020.

[33] A. Saini, M. S. Gaur, V. Laxmi, and M. Conti. Colluding browser extension attack on user privacy and its implication for web browsers. *Computers & Security*, 63:14 – 28, 2016.

[34] I. Sánchez-Rola, I. Santos, and D. Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *USENIX Sec*, 2017.

[35] D. Silver, S. Jana, D. Boneh, E. Chen, and C. Jackson. Password managers: Attacks and defenses. In *USENIX Sec*, pages 449–464, 2014.

[36] A. Sjösten, S. V. Acker, P. Picazo-Sanchez, and A. Sabelfeld. Latex gloves: Protecting browser extensions from probing and revelation attacks. In *NDSS*, 2019.

[37] D. F. Somé. Empoweb: Empowering web applications with browser extensions. In *S&P*, pages 227–245, 2019.

[38] O. Starov, P. Laperdrix, A. Kapravelos, and N. Nikiforakis. Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *WWW*, page 3244–3250, 2019.

[39] O. Starov and N. Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *WWW*, pages 1481–1490, 2017.

[40] O. Starov and N. Nikiforakis. XHOUND: Quantifying the fingerprintability of browser extensions. In *S&P*, pages 941–956, 2017.

[41] M. Ter Louw, J. S. Lim, and V. N. Venkatakrishnan. Extensible web browser security. In *DIMVA*, pages 1–19, 2007.

[42] usmedicalit. Another chrome extension is stealing passwords. `https://www.usmedicalit.com/2018/09/18/another-chrome-extension-is-stealing-passwords/`, 2020.

[43] vabr@chromium.org. Issue 636425: Value of autofilled input[type="password"] shows in dom as empty. `https://bugs.chromium.org/p/chromium/issues/detail?id=636425/`, 2016.

[44] M. Xie, J. Fu, J. He, C. Luo, and G. Peng. Jtaint: Finding privacy-leakage in chrome extensions. In *ACISP*, pages 563–583, 2020.

E

No Signal Left to Chance: Driving Browser Extension Analysis by Download Patterns

Abstract. Browser extensions are popular small applications that allow users to enrich their browsing experience. Yet browser extensions pose security concerns because they can leak user data and maliciously act on behalf of the user. Because malicious behavior can manifest dynamically, detecting malicious extensions remains a challenge for the research community, browser vendors, and web application developers. This paper identifies download patterns as a useful signal for analyzing browser extensions. We leverage machine learning for clustering extensions based on their download patterns, confirming at a large scale that many extensions follow strikingly similar download patterns. Our key insight is that the download pattern signal can be used for identifying malicious extensions. To this end, we present a novel technique to detect malicious extensions based on the public number of downloads in the Chrome Web Store. This technique fruitfully combines machine learning with security analysis, showing that the download patterns signal can be used to both directly spot malicious extensions and as input to subsequent analysis of suspicious extensions. We demonstrate the benefits of our approach on a dataset from a daily crawl of the Web Store over 6 months to track the number of downloads. We find 135 clusters and identify 61 of them to have at least 80% malicious extensions. We train our classifier and run it on a test set of 1,212 currently active extensions in the Web Store successfully detects 326 extensions as malicious solely based on downloads. Driven by the download pattern signal, our code similarity analysis further reveals 6,579 malicious extensions.

E.1 Introduction

Browser extensions are popular small web applications that users install in modern browsers to enrich the user experience on the web. Google's official extension repository, Chrome Web Store, currently has more than 180,000 extensions between browser extensions, apps, and themes, with many extensions having millions of users. Driven by the popularity of Chrome extensions, browser extension ecosystems have been adopted not only by Chromium-based browsers like Opera, Brave, and Microsoft Edge but also by browsers like Firefox and Safari. The latter browsers

draw on the same architecture, allowing developers to easily export their Chrome extensions. When an extension is installed, the browser typically pops up a message showing the permissions this new extension requests and, upon user approval, the extension is then installed and integrated within the browser.

The benefits of using browser extensions come at the high price of granting access to a vast amount of sensitive information. Not only extensions can get and interact with all the content of the web pages that users access but also if the extension defines the corresponding permissions, it can run some of the restricted APIs the browser exposes to extensions and retrieve sensitive information such as cookies, history and even modify the network traffic without the user knowledge. This raises serious security and privacy concerns [53, 54, 70].

Chrome Web Store. Extensions are usually stored in private repositories managed by vendors, where extensions developers upload them to be freely distributed afterward. The most popular browser extensions repository is the Web Store governed by Google, which banned the possibility of manually installing browser extensions from other sites different than the Web Store years ago [12].

The Web Store implements a Collaborative Filtering Recommendation System (CFRS) [25] in such a way that extensions are ranked or featured to make it easier for users to find high-quality content. This ranking is performed by a heuristic that takes into account ratings from users as well as usage statistics such as the number of downloads and uninstalls over time.

Inherent to CFRS, attackers have always been trying to promote or demote apps by automatically modifying ratings and raters [10, 43], or faking the downloads [8, 18]. Also, the proliferation of crowdsourcing sites like Zeerk, Peopleperhour, Freelancer, Upwork, and Facebook groups, have helped on this matter [42]. Among other things, by boosting some apps, developers may get funding from venture capitalists when their apps are popular among users [24].

The Web Store implements a set of fraud detection and defense mechanisms so that attackers cannot alter the ranking that easily [43]. Similar to Android Google Play, users can only review and rate an extension only if they 1) are logged in the Web Store, and; 2) install it first, being easier for Google to detect fake users trying to exploit the CFRS. However, this is not the case with downloads. To download and install extensions, users need a Chromium-based browser, e.g., Chromium, Chrome, and Brave. Therefore, the number of downloads can be easily altered by automatic processes, being difficult to differentiate between real users and automatic downloads. In this paper, we are particularly interested in how the downloads of the extensions can be used for grouping browser extensions based on the download patterns as well as identifying malicious ones based on such patterns.

Extensions' Downloads. We monitored the number of downloads of browser extensions over 6 months. We observe that the function defining the number of downloads is monotonically increasing over time for most extensions. However, there is a remarkable number of extensions whose downloads: i) do not follow an organic download pattern, i.e., they are synced with others, following the same pattern; ii) experience many fluctuations thus not following a monotonic function, and; iii) de-

viate from the usual growing pattern, i.e., they grow and/or decrease various orders of magnitude within two or three days. This leads to the insight that the number of downloads of the extensions can be leveraged as a useful signal for analyzing browser extensions. Based on these observations, we pose three research questions:

RQ1: Are there extensions that follow similar download patterns?

RQ2: Is there any relationship between download patterns and malicious code?

RQ3: Can we find malicious extensions based on their download patterns?

To answer these questions, we crawled the Web Store daily for 171 days and analyzed the download patterns of over 160,000 extensions. We clustered the extensions with respect to such patterns and found 135 clusters. Later, we analyzed the security of the extensions that compose these clusters and identify 61 of them to have at least 80% malicious extensions. Using a supervised learning algorithm, we trained two classifiers and evaluated them against 1 212 currently active extensions in the Web Store. The first classifier predicts which cluster in the training set the test set extensions are in. Afterward, a threshold is used to mark all extensions in a cluster as either malicious or benign based on the fraction of malicious extensions in the cluster. The second classifier instead directly predicts if an extension is either malicious or benign. The first classifier successfully detects 326 and the second detects 289 extensions as malicious solely based on downloads. Finally, driven by the download pattern signal our code similarity analysis results in discovering 6,579 malicious extensions.

Contributions. In detail, our contributions are:

1. We describe the methodology we use to retrieve and analyze the data from the Web Store during 171 days (see Section E.3);
2. We show our results, supporting that *RQ1)* Extensions follow similar patterns. *RQ2)* These patterns can be correlated to maliciousness *RQ3)* We can find new active malicious extensions based on the download patterns. (see Section E.4);
3. We present a real example of 29 extensions whose downloads are all synced and hijack the search queries of the users, which leads us to discover 6,579 extra hijacker extensions that remain hidden in the Store (see Section E.5).

We introduce some basic definitions for an easy understanding of the paper in Section E.2, discuss the threats to validity in Section E.6, offer a summary of the most relevant related work in Section E.7 and conclude the paper in Section E.8.

Coordinated disclosure. We reported to Google 6,579 the malicious extensions detected in our empirical study as well as our methodology. The Chrome Web Store team removed 4,858, while 1,721 are still under investigation.

Artifacts. We open-source our code and data needed to reproduce the results presented in this paper here: `https://www.dropbox.com/sh/clde4ui89qkdc72/AACeoirHy9WcASjrV9s1yhEta?dl=0`.

E.2 Preliminaries

In this section, we summarize the security and privacy threats that extensions pose, some basic concepts of time-series, some definitions we use in the paper and introduce the threat model.

E.2.1 Browser Extensions' Security & Privacy

Browser extensions are small applications that can help both developers and users while developing new web applications or surfing the Internet. However, due to the amount of sensitive information the extensions have access to when they run in the users' browser, the security and privacy of such data have been cast doubt on the adoption of the extensions.

Extensions are composed of two main parts, content scripts, and background pages. The former are JavaScripts automatically injected in the web pages the extension defines in the manifest file under the `content_scripts` key. On the other hand, background pages are JavaScripts with no direct access to the web content but with access to a set of restricted and privileged set of APIs the browser exposes, e.g., network traffic, cookies, and history. To access these APIs, the extension has to define in the manifest file the permissions associated with every API it attempts to use.

E.2.2 Time-Series Analysis

A time series is a set of data points ordered by time. There are different metrics to compare two time series, i.e., to get how similar they are, usually based on the distance among the data points of the series. The most common examples are the Euclidean distance, the Longest Common Subsequence (LCS), and the Dynamic Time Warping (DTW), being this last one the most common distance used to compare time-series [17]. In addition to that, Canonical Time Warping (CTW) is a method based on Dynamic Time Warping (DTW) that aligns time series under rigid registration of the feature space, not being needed that time series share the same size nor the same dimension.

Learning in Machine Learning (ML) can be classified into two main families: supervised and unsupervised. Supervised learning needs labeled data to learn the mapping function from an input to an output whereas in unsupervised learning algorithms there is no labeled data and, therefore, they learn patterns from the input data. Research in both supervised and unsupervised learning algorithms applied to time series is quite active nowadays.

Clustering is an unsupervised learning technique by which similar data are grouped with little or no knowledge in advance about the data. Time-series clustering is a particular case of clustering where the series, a large number of points measured chronologically, are handled as single objects to extract patterns among them [3]. Examples of algorithms used for time-series clustering are Self-Organizing Map (SOM) [3], k-means [27], k-shapes [38], and shapelets [67].

Classification is a supervised learning technique by which an algorithm analyzes a training dataset and outputs function used for determining the labels of new examples. Some examples of algorithms for time-series classification are KNeighbors [56], Support Vector Machines (SVM) [29], Rocket [15], and Minirocket [16].

E.2.3 Definitions

In the following, we explain in detail each one of the key concepts used throughout this paper.

Downloads. The number of downloads the Web Store publicly exposes for every extension (e) at time t.

Increment of downloads ($\Delta_{\mathbf{d}_e}$). Is the difference, in absolute value, of two consecutive downloads of an extension e, i.e., $\Delta_{\mathrm{d}_e} = |t_i - t_{i+1}|$.

Average of the increment ($\overline{\Delta_{\mathbf{d}_e}}$). Is the arithmetic mean of the increment of the downloads, i.e., $\overline{\Delta_{\mathrm{d}_e}} = \frac{\sum_{i=1}^{n} \Delta_{\mathrm{d}_e}}{n}$, where $i < n$ and n is the number of measurements per extension.

Average of the average of the increment ($\overline{\overline{\Delta}}$). Is the arithmetic mean of all the average of the increment of the downloads of the extensions, i.e., $\overline{\overline{\Delta}} = \frac{\sum_{i=1}^{e} \overline{\Delta_{\mathrm{d}_i}}}{n}$, where $i <= n$, n is the number of extensions and $\overline{\Delta_{d_i}}$ is the average of the increment of an extension i.

E.2.4 Threat Model

Extensions can pose many threats to users' privacy and security. Previous works have analyzed extensions that inject adware, track and fingerprint users, takeover search engines, modify security headers, execute remote code, persuade and steal user's search queries [4, 11, 20, 33, 37, 48, 49, 50, 52, 53, 54].

In this work, we focus on a subset of these attacks, namely the search query stealing attack. This is prominently used by "Wallpaper" extensions user's search queries [20]. These extensions override the new tab functionality of the browser such that when the user opens a new tab, this is replaced by the one the extension provides. They usually provide a search bar with some arbitrary wallpaper backgrounds. However, these extensions can redirect search requests containing sensitive queries and redirect them to different URLs (see Table E.5). By focusing on a subset of all possible attacks, we can more accurately perform the security analysis that serves as the ground truth for the time-series analysis.

E.3 Scrutinizing the Web Store

A Collaborative Filtering Recommendation System (CFRS) is a system that keeps track of the users' preferences to use it afterward to offer new suggestions to other

users [35]. Youtube, Amazon, and Netflix are examples of applications that implement CFRSs [45, 64]. This is also the case of the Google Web Store, the online marketplace where browser extensions are freely distributed. The Web Store implements a CFRS where extensions are ranked based on parameters like ratings from users, number of downloads and uninstalls over time among others.

Even though the algorithms used by the CFRS are usually unknown, researchers found attacks against the recommendation system, being pollution attacks the most common ones [23, 45, 64]. Such attacks consist of generating fake data typically in form of new users that interact with the system either watching videos, reading books, and rating or downloading items. By doing so, attackers may promote or demote items as desired.

Some of the information the Web Store offers for each browser extension is the category it belongs to, the name of the developer, the company, a general description, some privacy practices, users' reviews, the number of downloads, the rates that users give or metadata like the version of the extension, and when it was updated.

In this section, we present the methodology we follow to identify download patterns as a useful signal for analyzing browser extensions. We leverage machine learning for clustering extensions based on these patterns, confirming at a large scale that many extensions follow strikingly similar download patterns (see Section E.4).

We split our methodology into four main tasks (see Figure E.1):

Data Gathering Daily monitoring of the Web Store to extract downloads of all the browser extensions;

Security Analysis We combine static, manual, and dynamic analysis to mark malicious extensions.

Time-Series Analysis Firstly, group extensions according to the downloads function they describe (based on Δ_{d_e}) and look for patterns (clustering phase) and label them based on the security analysis. Secondly, we implement a learning algorithm based on the downloads, and;

Discovering We rely on *DeDup.js*, a recently published framework [39] to discover extensions shipping the same (malicious) files detected during our security analysis.

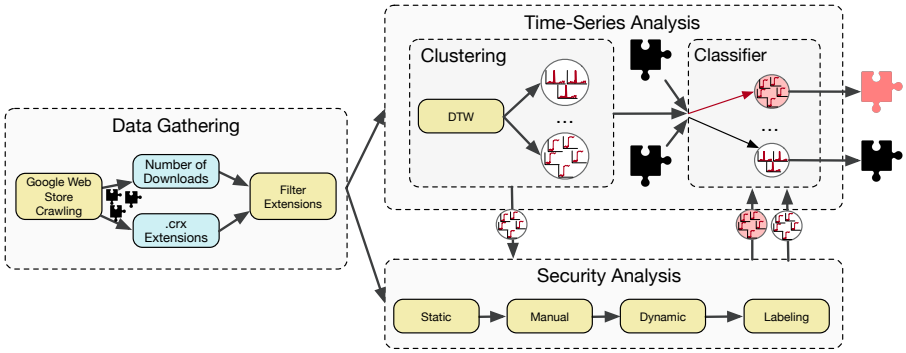


Figure E.1: Systematic methodology to cluster and extensions.

E.3.1 Data Gathering

We observe that Google makes the number of downloads public for most of the extensions but not for all of them. At the time of writing, there are 10,941 extensions whose downloads are not shown on the page. Unfortunately, we do not know why some are hidden. Even though Google hides the downloads for some extensions, we found out that such a number is still in the source code of each browser extension but commented under the <PageMap> HTML label, and more concretely it comes as a text field under the <Attribute> label whose attribute name is "user_count" (see Listing E.1).

```
<!--<PageMap>
  <DataObject type="document">
    ...
    <Attribute name="user_count">
    6281
    </Attribute>
    ...
  </DataObject>
</PageMap>-->
```

Listing E.1: Number of downloads hidden in the Web Store's HTML source.

Between March 2021 and Aug 2021, we crawl the Web Store daily, monitoring the downloads of all the extensions and their version. From all the download patterns we extract all the extensions whose $\overline{\Delta_{d_e}}$ is larger than $\overline{\overline{\Delta}}$. These are the extensions whose downloads fluctuate more than the global average in the store.

Dataset Filtering. After the extraction of all the public data of the extensions, we filtered the dataset in terms of size (number of extensions) and data information (number of measurements) allowing us to perform a more accurate security analysis in Section E.3.2 and better clustering in Section E.3.3. This filtering process neither affects the results nor the methodology presented in this paper. With more manual effort, in terms of time spent on security analysis and clustering, and increasing the frequency of the data gathering, we believe this method can be extended to any group of extensions and attacks.

First, we focused on a subset of extensions called wallpapers, i.e., browser extensions that override the starting page the user previously had and replace it with a random background image that changes every time the user refreshes the webpage and a search box in the middle of the screen. We did so because researchers recently showed that many extensions attempt to steal users' search queries [20]. Wallpapers can be found in the 11 categories of the Web Store. To get all the wallpapers, we filtered out extensions that do not define `chrome_url_overrides` in their manifests. Second, we analyzed those extensions with more than 90 measurements which, given our crawling frequency, corresponds to at least 90 days.

E.3.2 Security Analysis

In accordance with our second research question *Is there any relationship between download patterns and malicious code?* we need to perform a security analysis of the extensions to label them as malicious or benign. Using these security labels we can search for relationships between clusters and malicious code.

While there are many possible attacks malicious extensions can perform, we focus on malicious extensions that change the user's search engine without notice or steal their queries. This can be accomplished either by redirecting the search or using analytics. To detect this we develop a fully automatic dynamic analysis method (see Section E.3.2.1), and verify it using a combination of static and manual analysis (see Section E.3.2.2).

E.3.2.1 Dynamic Analysis

To detect if an extension steals search queries, we use dynamic analysis to interact with the extension. The goal of this interaction is to trigger a search if the extension has a search function.

Challenge. The challenge is that this search function can be implemented in many different ways. For example, simple HTML forms with a predefined action URL for the search engine could be used. In this case, a simple static analysis would likely be enough. However, the search bar could instead be a dynamically generated text-field with JavaScript events connected to complex and obfuscated frameworks, making static analysis more difficult. Additionally, extension can use iframes or redirect users to other new tab pages where the search stealing takes place.

The query stealing can also be implemented in different ways. Normally, server-side redirects are used to send users and their queries to other servers before arriving at the intended search engine. For example, when searching, the query is sent to `newtab.com?query=secret` which then redirects it to `google.com?query=secret`. Another method frequently used is JavaScript analytic scripts that use `XHR` to send the queries to an analytic platform before sending the user to the intended search engine. Analyzing analytic frameworks statically is also challenging, which is why we believe the dynamic analysis is better suited.

Orthogonal to the complexity of statically scanning code, extensions can also implement delays before they start misbehaving. If these delays are implemented in the extension's source code it could be detected statically. However, if instead

extensions rely on external websites for the new tab features, these servers could implement the time delay, making it impossible to detect statically.

To tackle these challenges we divide our dynamic analysis into two phases. The first one scans the extensions while keeping track of any search queries being stolen. Also, it records if the extensions rely on external websites for the functionality. If the first phase does not detect the extension as malicious and it is using a website, then, in the second phase, we scan the website for a prolonged period. Using a long enough time span we increase the chances to catch the switch from benign to malicious behavior.

Phase 1 - Scanning extensions. To detect search query stealing, we use Puppeteer[1] to control a Chrome instance running with a potentially malicious extension installed. We build our solution on the dynamic analysis infrastructure developed by Eriksson *et al.* [20]. However, our approach to interact with the extensions are markedly different. Compared to previous approaches that tried to interact with extensions using honey pages or visiting different websites, our approach is more akin to web crawling and web vulnerability scanning. Honey pages are good at detecting how extensions interact with web pages; however, we want to interact with a web page generated by the extension, i.e., the new tab page.

By using Puppeteer we can mimic how a user would interact with the extension by clicking on search fields and typing queries with their keyboards. This is an important difference from using JavaScript to change the values of search fields as that might not trigger events and analytic scripts.

In Figure E.2, we give an overview of the algorithm we use to interact with the extension. As some extensions load their new tab functionality in an iframe we need to ensure that we check those first using the same method. We perform two core interactions: 1) First clicking on each text input of either type `text` or `search`. Then we type a predefined query, i.e., "Secret00133700Query". This will trigger any event listeners waiting for a user to type their query; 2) We once again click on each element but this time simulating pressing enter on the keyboard. This submits the search query.

[1] https://github.com/puppeteer/puppeteer

Algorithm 1 Dynamically scan extensions for query stealing

```
1:  procedure SCAN(extension, query)
2:      page ← loadNewtab(extension)
3:      for f ∈ getIframes(page) do
4:          for i ∈ getInputs(f) do
5:              Click on i
6:              Type query
7:          end for
8:          for i ∈ getInputs(f) do
9:              Click on i
10:             Press Enter
11:         end for
12:     end for
13:     for i ∈ getInputs(page) do
14:         Click on i
15:         Type query
16:     end for
17:     for i ∈ getInputs(page) do
18:         Click on i
19:         Press Enter
20:     end for
21: end procedure
```

Figure E.2: Dynamically scan extensions for query stealing

After the interaction method we check all the network traffic to detect any requests containing our query. Similar to previous work [47], we also check for transformations on the query, e.g., lowercase, uppercase, BASE64 etc. If we detect the query in a request to or from a server that is not one of the four major search engines (Google, Yahoo, Bing, and DuckDuckGo), we mark it as malicious.

Phase 2 - Scanning websites As some extensions rely on websites for their new tab functionality, we also scan these websites in an attempt to detect delayed malicious behavior. For example, a new tab website can start with using Google as the search engine but after some time start redirecting the search queries.

To find websites worth scanning we analyze the results from *Phase 1*. We extract all pairs of *source* and *target* URLs where the target contains our search query. For example, when analyzing an extension we found `tabhd.com` as the source and `google.com/?q=QUERY` as the target. In this case, we scanned `tabhd.com` to test if it switches from `google.com` to something else.

For each website we find in *Phase 1* we pick an extension relying on this website and scan it repeatedly using the setup in *Phase 1*. We reload the new tab page every 30 seconds for an hour to detect any changes where search queries are sent. If we find that a website starts redirecting queries after some time, we mark the website as malicious and all extensions which use the website.

E.3.2.2 Verification

While it is unlikely that the dynamic analysis results in false positives we still manually test some of the extensions to detect any false negatives. Note that false negatives found in this verification step do not affect the final results we present, i.e., the security labels we use in later stages for clustering and classification are based on the output from the dynamic analysis. The reason for false positives being unlikely is that our dynamic analysis only marks an extension as malicious if it detects a request from an unknown server with our query. To better understand the general limitations of our dynamic analysis and approximate the false negatives we perform a manual analysis of extensions marked as benign and use static analysis to find similar ones.

Manual From the set of extensions marked as benign by the dynamic analysis, we pick a subset to inspect manually. To do so, we open it in Google Chrome and follow any installation guide the extension needs. We also accept any pop-ups, either modal windows or window pop-ups that might show up during this process. From there we locate the search bar if one exists. Using the network tab in developers tools we note all requests being sent both when typing in the search bar and when submitting the query. If any unknown server receives our query we mark the extension as malicious.

Static To increase our coverage, we include static analysis to automatically find similar extensions to the ones we find in the manual analysis. We achieve this by manually creating simple signatures for each new malicious extension we find. For example, Listing E.2 shows a code snippet that could be a signature for a group of extensions:

```
1 var url_analytics = 'https://s.example.com'
```

Listing E.2: Example of code used as a signature to find more similar extensions.

Using these signatures we search all manifests, HTML, and JavaScript files for the signature. If we find a match we mark these as malicious.

E.3.3 Time-Series Analysis

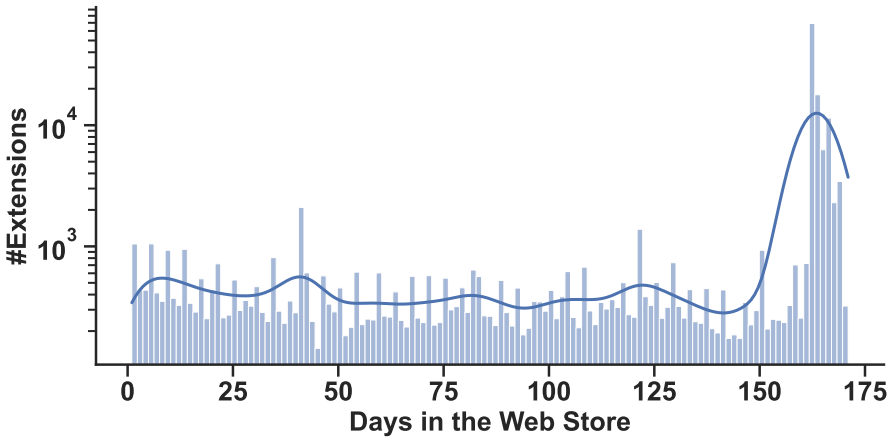
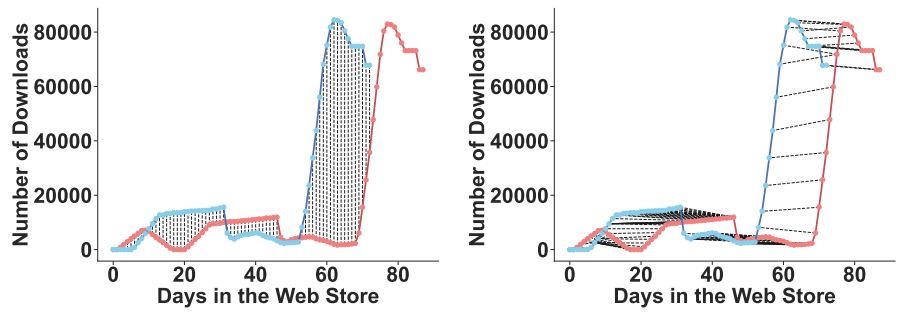


Figure E.3: Number of days (x-axis) monitoring the extensions (y-axis in log scale).

Ideally, after 171 days of daily monitoring, we should have gathered 171 measurements per extension. However, extensions can be deleted or added, thus affecting the number of downloads collected per extension. We show in Figure E.3 the distribution of how long the extensions in our dataset were online in the Web Store.



(a) Euclidean distance **(b)** DTW distance

Figure E.4: DTW vs Euclidean distance applied to extensions downloads time-series.

Clustering To answer our first research question (**RQ1**): *Are there extensions that follow similar download patterns?* We apply state-of-the-art clustering methods. The results from this clustering will illuminate if there are any clusterable download patterns.

Given the heterogeneous distribution of the data, i.e., extensions are alive in the Web Store for different periods, we could not implement classical clustering method-

ologies based on euclidean distances like DBSCAN or other statistical values such as mean like K-means. The reason is that to compare two time-series using the euclidean distance, both series need exactly the same amount of data as well as being synchronized (see Figure E.4a). Instead, we adopted a well-known technique in time-series clustering named Dynamic Time Warping (DTW) that solves the aforementioned constraints by computing a discrete matching between the elements of both series rather than using their time sequence [3] (see Figure E.4b).

In this paper, we follow a so-called Human-in-the-Loop (HitL) methodology [63] combined with DTW to cluster time-series downloads of browser extensions. To do so, we deploy an instance of dtadistance library [34] combined with COBRAS-TS [60], an interactive version of COBRA [13] that allows semi-supervised clustering of time series. However, this process can be fully automatized without including any human in the clustering algorithm.

Classification To answer our third research question: *Can we find malicious extensions based on their download patterns?* We create two classifiers that aim to classify extensions as malicious solely based on download patterns. The first one classifies directly based on download patterns while the second cluster similar patterns before classifying each cluster as malicious or benign. We implement and evaluate an instance of MiniRocket [16]We split our extensions dataset into training and test sets. To simulate a realistic scenario of our approach, we use all extensions that had been deleted at the end of the data-gathering phase as the training set and the still-active ones as our test set.

We implement two different classifiers, both using MiniRocket. The first one predicts if the extension is malicious based on the download patterns and security labels from our security analysis. The second predicts which cluster from the training set the extensions in the test set are closest to. Finally, we compare a threshold t to the fraction of malicious extensions in the cluster. If t is greater than this fraction we mark the extension as malicious.

We evaluate our model according to three main metrics: precision, recall, and F1-Score. Precision measures how many positive predictions are true, i.e., $TP/(TP+FP)$. Recall measures how many positive classes the model can predict, i.e., $TP/(TP+FN)$. Finally, F1-Score is the harmonic mean of both recall and precision, i.e., $2(\text{recall} \cdot \text{precision})/(\text{recall} + \text{precision})$.

E.3.4 Discovering

Discovering malicious extensions is challenging given the dynamic nature of JavaScript and the complexity of the browser extensions ecosystem together with the extension's use of remote servers for full or partial functionality. In this paper, once we detect malicious behavior in an extension, we implement two strategies to discover as many related malicious extensions as possible.

Firstly, we statically analyze the manifest looking for similarities in how extensions define and use the files they are composed of. With this, we identify the file that performs the attack. Secondly, since we are not interested in similarities in the structure but in the exact files, we contacted the authors of *DeDup.js* [39], a frame-

work for discovering potentially malicious extensions by leveraging deduplication, and shared both the source code and the dataset with us. DeDup.js maintains an updated dataset with the Subresource Integrity (SRI) of all the static content of the extensions (CSS, fonts, HTML, images, and JavaScript). Thus, when we detect a malicious extension, we isolate its files and detect which JavaScript files are responsible for the malicious behavior, and compute its SRI. Later, we use DeDup.js to discover how many extensions contain the same malicious file and automatically certify that such a file is used in the same way as the malicious extension we initially detected.

E.4 Results

In this section, we present the results from our data gathering, security, and time-series analysis. We use these results to answer our research questions.

E.4.1 Data Gathering

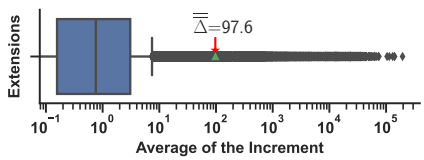


Figure E.5: Distribution of the average of the increment of downloads ($\overline{\Delta_d}$). The $\overline{\Delta_d}$ of 8,165 extensions is higher than the average ($\overline{\overline{\Delta}}$).

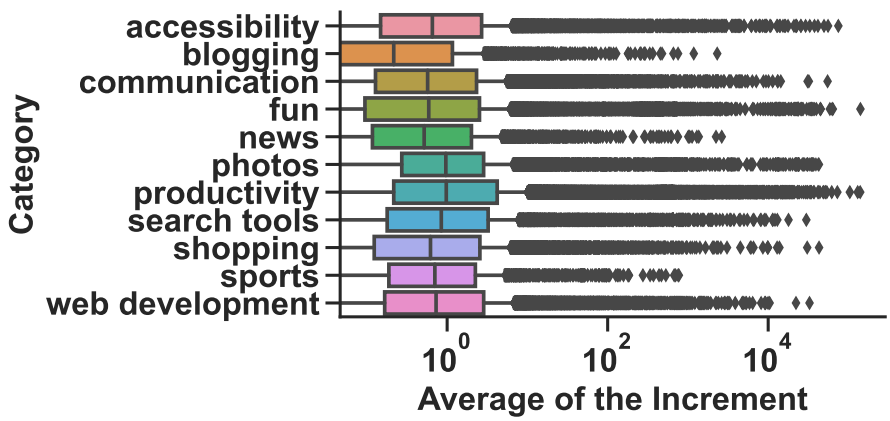


Figure E.6: Distribution of the average of the increment of downloads group by the 11 categories of the Web Store.

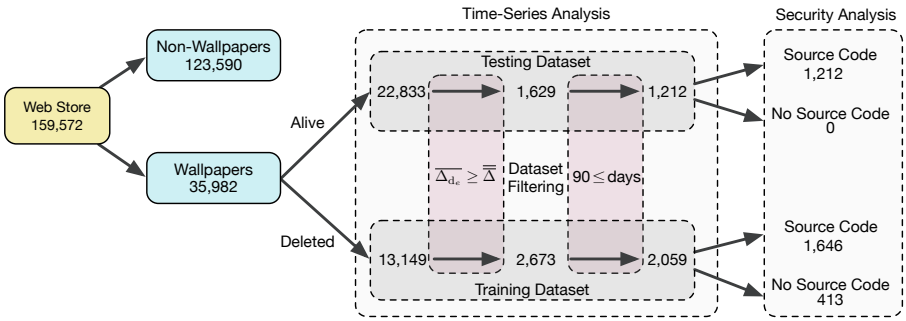


Figure E.7: Filtering Process. Extensions on every step.

In total, after 171 days of monitoring, we collected download patterns for 159,572 extensions. In Figure E.5, we show the distribution of the average of the increments of the downloads ($\overline{\Delta_{d_e}}$) of all the extensions of the Web Store. Interestingly, we can see that there are many outliers, i.e., extensions whose $\overline{\Delta_{d_e}}$ is higher than 10, 100, 1,000, or even 10,000 downloads. We marked with a green triangle in Figure E.5 the average of $\overline{\overline{\Delta}}$ of the extensions (around 97.6). Even though we could have analyzed all the extensions of the Web Store, we restricted ourselves to the 8,165 outliers extensions whose average of $\overline{\Delta_{d_e}} \geq 97.6$.

In Figure E.6 we split the dataset into the categories the extensions belong to. We also extracted the last public download the Web Store offered per extension and include in Figure E.11 (Section E.I) the download distribution of extensions split into categories. Although there are some extensions with millions of downloads, in general, we can observe that most of the browser extensions have been downloaded less than a hundred times, with even fewer downloads in some categories, including "blogging".

In summary (see Figure E.7), we collect 159k download patterns from the Web Store. From these, 35k are wallpaper extensions, where 22k are still active, and 13k are deleted. From them, we first filtered extensions with interesting download patterns, (i.e., $\overline{\Delta_{d_e}} \geq 97.6$), getting 1 629 and 2 673 still-active and deleted wallpapers respectively. Finally, because of our crawling frequency (once a day), to increase the useful information of the downloads and thus reduce the false positives, we analyze the downloads of the extensions that remained in the Web Store longer than 90 days, resulting in a total of 1 212 and 2 059 alive and deleted extensions. We use these 3,271 wallpaper extensions for security analysis and clustering.

E.4.2 Security Analysis

In this section, we present the results of our automatic security analysis where we find 1,292 malicious extensions. Further, we explore how these malicious extensions related to the clusters of download patterns and answer our second research question: *Is there any relationship between download patterns and malicious code?*

Table E.1: Popular domains used by query stealing extensions.

Domain	#Extensions
cse.google.com	146
mc.yandex.ru	134
gundil.com	116
cors-anywhere.herokuapp.com	100
www.google-analytics.com	92
completion.amazon.com	60
s.bingparachute.com	42
addiyos.com	16
the-theme-factory.com	14
chromethemesonline.net	11

Phase 1 - Scanning extensions We dynamically executed and analyzed 2,858 extensions, which is the number of extensions we had the source code for (see Figure E.7). For the remaining 413 extensions, we marked them as benign, since we can not prove they are malicious. It takes approximately 30 seconds to analyze one extension, and our code runs 8 extensions in parallel, resulting in a total of about 3 hours to analyze all our extensions.

In total, we found 441 malicious extensions stealing search queries during the first phase of our dynamic analysis. They use a combined total of 182 different domains for their query stealing. However, one extension can use multiple domains, e.g., one extension[2] uses `search.myway.com` for searching while simultaneously using Google Analytics to log the query.

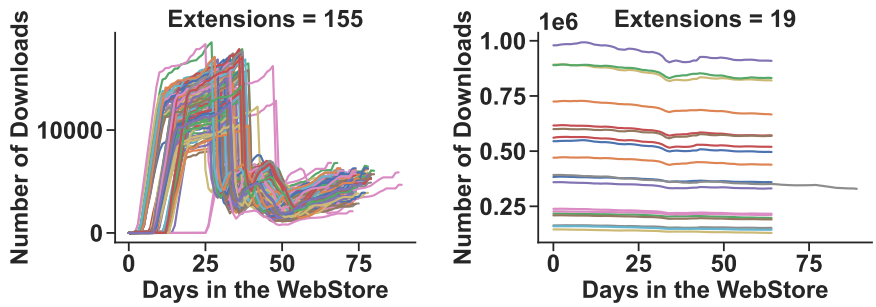
In Table E.1, we present the ten most used domains. Note that these domains themselves are not necessarily malicious but are being used by malicious extensions. For example, `cse.google.com` is not malicious but is commonly used by spyware [14].

Phase 2 - Scanning websites We analyzed the websites used by non-malicious extensions from *Phase 1*. In total, we found three domains that were used by 852 extensions, we present these in Table E.2. We analyzed each for an hour and detected both `www.tabhd.com` and `www.ultitab.com` switch from benignly using Google to maliciously using `gundil.com`. `themes.wallpaperaddons.com` was consistently using Google and was therefore not marked as malicious. From this phase, we marked 851 additional extensions as malicious. In total from both phases, we found 1,292 malicious extensions.

[2] bcdhacjdengeibbbhmdjodiecaiciehc

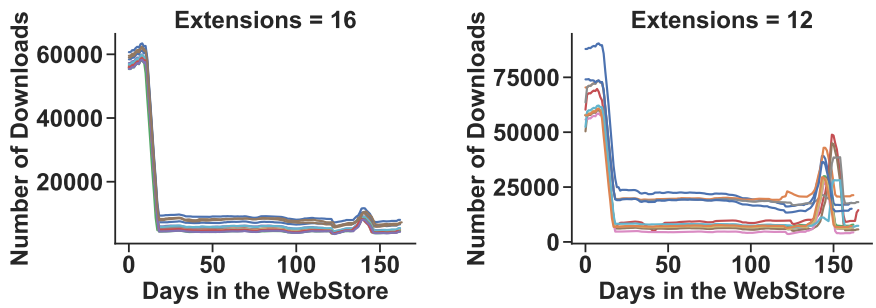
Table E.2: Domains scanned in Phase 2.

Domain	Malicious?	#Extensions
www.tabhd.com	Yes	667
www.ultitab.com	Yes	184
themes.wallpaperaddons.com	No	1



(a) TabHD extensions **(b)** MyWay extensions

Figure E.8: Download patterns for two malicious clusters.



(a) FreeAddon extensions **(b)** FreeAddon extensions

Figure E.9: Download patterns for two benign clusters.

Verification During our verification, we picked a sample of 100 benign extensions to manually review. As discussed in Section E.6 we are mainly interested in finding false negatives, therefore we only consider benign extensions for manual review. Most of these were correctly marked as benign, being developed by either *FreeAd-*

Table E.3: Distribution of malicious clusters versus the extensions they are composed of.

	Maliciousness					#Total
	0%	0%-50%	50%-80%	80%-100%	100%	
#Clusters	52	15	7	6	55	135
#Extensions	902	1130	299	401	539	3,271

don or *Choosetab*. We did find a total of three extensions we incorrectly marked as benign.

To better understand our limitations we look outside the sample for similar extensions to the ones we missed. Here we find a total of 32 extensions out the 2,858 extensions. The biggest miss was due to a group of 27 extensions using `s.tabturbo.com` to steal the queries.

There was another group of three extensions we failed to correctly mark as malicious due to a limitation of Puppeteer, and possibly a bug in the extension. Upon loading the extension, it uses the code in Listing E.3 to try to redirect the user to `extension://index.html`, on Ubuntu this causes Chrome to prompt the user to execute an external program. This prompt can not be closed by Puppeteer and at the same time makes it impossible to interact with the web page.

```
n = document.createElement("meta"),
n.httpEquiv = "refresh",
n.content = "0;url=extension://index.html",
document.getElementsByTagName("head")[0].appendChild(n),
```

Listing E.3: The extension creates a meta tag which prompts the user an external program. This prompt blocks our analysis from interacting with the extension.

Finally, there were two unique extensions with different problems. The first one had two search input forms, but only one malicious, and we only tested the benign one. The second one had a hidden search input which our dynamic approach failed to click on. We discuss these cases more in Section E.6.

E.4.3 Time-Series Analysis

Clustering. In this section, we present the results of our clustering and how it relates to our first research question (**RQ1**): *Are there extensions that follow similar download patterns?* After clustering the extensions based on their download patterns, as explained in Section E.3.3, we find a total of 135 clusters composed of 3,271 extensions, with an average of 24 extensions per cluster and 39 clusters with more than 10 extensions (see Table E.3). We show four examples of these clusters in Figures E.8 and E.9 whereas we include more examples in Section E.III.

Note that the cluster in Figure E.8b might seem inactive but a small drop, around day 30, corresponds to hundred thousands of downloads. Another example is the

cluster in Figure E.9b, where the extensions are inactive for almost 100 days before all gaining a lot of downloads simultaneously. These clear patterns confirm that there exist extensions that follow similar download patterns.

Malicious Clusters. To answer (**RQ2**): *Is there any relationship between download patterns and malicious code?* We further analyzed the 135 clusters and found that similar clusters have similar attack patterns. For example, in one cluster with 155 extensions, 124 used `tabhd.com` to steal queries (see Figure E.8a). Note that we did not have the source code of 30 out of the remaining 31 extensions in that cluster and therefore marked those as safe (we give more details in Section E.6). Using an unofficial repository [21] we could confirm that those 30 extensions were also using `tabhd.com`. While, in another cluster, 13 out of 19 extensions used `search.myway.com` instead (see Figure E.8b). In this case, we miss the code of 5 extensions. Similar to malicious clusters, Figure E.9a shows two clusters of 16 and 12 extensions being all benign, being all from FreeAddon.

Based on these results, we can confirm that there is a relationship between download patterns and malicious code in extension. Furthermore, in addition to a correlation between download patterns and maliciousness we also find some cases where the download pattern correlates directly with the specific details of the attack, for example, the use of `tabhd.com` as a website.

Classifier In this section, we present the results relating to our final research question (**RQ3**): *Can we find malicious extensions based on their download patterns?* First, we present the result of using the classifier to predict which cluster an extension matches then we present the fully automatic solution where security labels, i.e., malicious or benign, are directly predicted.

We train our classifier on download patterns for 2 059 extensions and use 1 212 as test. Our classifier will first match each extension in the test set with a cluster from the training set then, based on a threshold t, determine if the extension is malicious.

Figure E.10 depicts the ROC curve of our classifier. In it, we see how the threshold t affects the recall and precision of our classifier, achieving the best F1-Score of 0,89 at $t = 0.26$, with a precision and recall of 0,88 and 0,90. At this threshold, the classifier finds 326 out of the 364 malicious extensions in the test set demonstrating that we successfully find malicious extensions solely based on their download patterns.

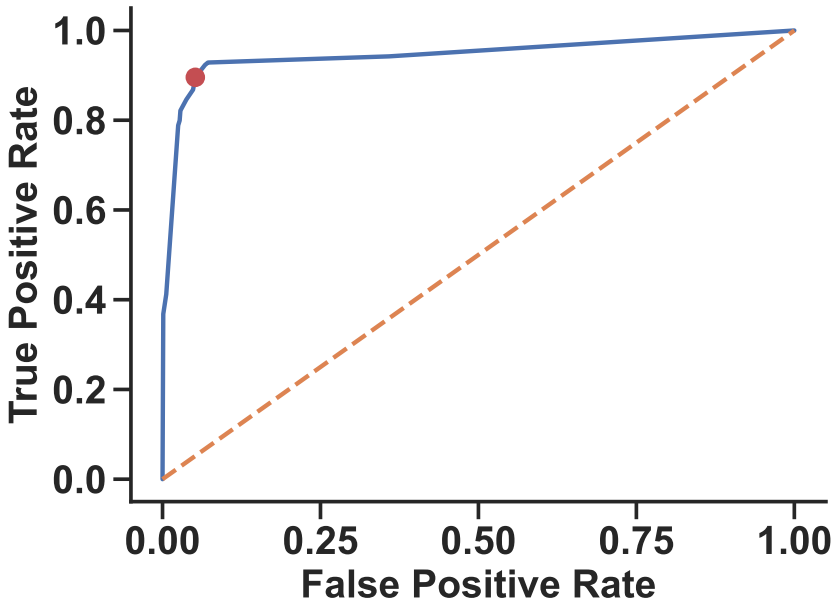


Figure E.10: ROC curve of our classifier. The red circle marks the best F1-Score (0,89), correctly classifying 326 out of 382 malicious extensions.

Directly predicting labels results in an F-score of 0.89 with a precision of 0.92 and recall of 0.80. This is slightly worse than predicting clusters with the optimal threshold. We believe this is because the semi-automatic clustering is better than the fully automatic solution used here, which translates into better security label classification. Solely predicting labels also fails to capitalize on the fact that there are cases where download patterns correlate with code or attack similarities.

E.5 Use Case: Search Hijacking Wallpapers

In this section, we perform an in-depth analysis of a group of active malicious extensions we found using our classifier. Furthermore, we use code similarity analysis to extend the number of malicious extensions we find, as explained in Section E.3.4.

Using our classifier, we discovered a cluster of 29 malicious extensions that when installed, redirect the user to websites whose look & feel are quite similar to the HTML that other extensions enclosed in the group of wallpapers provide (see Table E.4). The purpose of these extensions is to prove a search bar with randomly generated background images to the user each time she opens a new tab in the browser.

We further analyze the code and structure of these extensions. Initially, the extensions seem harmless as they do not ask for any permission nor have they any

Table E.4: Servers where the 29 extensions redirect users to.

URL	#Extensions
https://www.ultitab.com	19
https://www.tabhd.com	10

known harmful files. In Listing E.6, we include the manifest file of these 29 extensions, having all of them the same manifest but with different icon (`<iconFileName>`), background page (`"js/<name>.js"`) and version (`<version>`).

However, when we automatically installed them all, we realized that all of them overwrite the `"newtab"` property, automatically loading the output.html web page whose content is `<head><script src="js/main.js"></script><head>`, i.e., this files just loads the main.js file. Interestingly, such a JavaScript file just has one line: `document.location='<url>/<name>';` where `<url>` is the server URL and `<name>` is usually the name of the extension.

When accessing either tabhd.com or ultitab.com, the servers automatically generate a random name and redirect users to https://<url>/<name> where `<name>` is the name of a random wallpaper extension. The content of all the URLs is the same, i.e., there is a search bar in the center and some links to the privacy policy, settings, or user agreement.

The key part is the search bar that automatically redirects the search queries of the user to Google during the first 30 minutes. After that, the web server provides almost the same content but the queries the users introduce in the search bar are redirected to https://gundil.com/index.html?q=<query> instead. Such a website looks identical to Google but the first 4 entries are always advertisements. We also certify that this website tracks users by retrieving information such as the IP address, browser type, browser version, the pages that users visit, the time and date as well as the time spent on those pages, unique device identifiers, and other diagnostic data.

E.5.1 Wallpapers Discovering

To discover more malicious extensions utilizing the same attack, we statically analyzed all the wallpapers (i.e., 35,982) stored in the Web Store looking for extensions that, similar to the analyzed ones: 1) implement `chrome.browserAction.onClicked.addListener()` and `chrome.runtime.onInstalled.addListener()` event listeners and the `chrome.tabs.create()` function in them; 2) add the `newtab` property of the `chrome_url_overrides`, and; 3) include any URL in their JavaScripts.

We classified extensions into 38 groups according to the URL they include and visited each one of the websites as stated in Section E.3.2. The look & feel of all the 38 sites were similar: apart from having some icons to social networks and some other shortcuts, all of them provide a search input similar to Google and Bing pages.

The most used URL by the extensions is https://mytab.me/<alias> where <alias> is a string that can be found in the app.js file. As an example, the app.js of the "Burst Gyro Anime Anime New Tags Hot HD Themes"[3] extension is `window.app={domain: 'https://mytab.me', name: 'anime1–6'};` and the redirection is carried out in the index.js (`(() => {location.href = `${app.domain}/${app.name}/`;})();`), script that is loaded by the HTML file that overwrites the `"newtab"` property of the browser. Hence, redirecting the user to an external server every time she opens a new tab. In Listings E.4 and E.5 we present the background and content script that 2,554 and 3,637 extensions using https://mytab.me have in common respectively. The background script of the remaining 1,083 extensions is the same but without the last `chrome.runtime.setUninstallURL()` statement.

```
(() => {
  chrome.browserAction.onClicked.addListener(
    function() {
      chrome.tabs.create({
        url: `${app.domain}/${app.name}/`
      });
    });
  chrome.runtime.onInstalled.addListener(
    function(details) {
      if (details.reason == "install") {
          chrome.tabs.create({});
      }}
  );
  chrome.runtime.setUninstallURL('https://mytab.me/?from=
    uninstall')
})();
```

Listing E.4: Background script of 2,554 sleeping malicious extensions.

```
(() => {
  location.href = `${app.domain}/${app.name}/`;
})();
```

Listing E.5: Content script of 3,637 sleeping malicious extensions.

We realized that some of the web pages initially use a legitimate search engine like Google or Bing but after some time (between 1 to 30 minutes) they redirect the search queries to other search engines (e.g., https://gundil.com/index.html?q=<query> and https://str-search.com/results.php?q=<query>) similar to what the extensions we detected by analyzing the downloads, extract personal metadata from users to fingerprint them as well as adding custom ads.

In addition to that, a few of them perform a double redirection, i.e., they first redirect the query to an external server and redirect the user later to a legitimate search engine, being almost unnoticeable to the user. Let us give two examples, one using Google Chrome and another one using Bing.

[3] kidpaijejhfcmlceagcmkmcpbfmclhfi

Table E.5: Top 10 of the most used URL by extensions to hijack search from users.

URL	#Extensions
https://mytab.me	3,637
https://www.tabhd.com	857
https://www.ultitab.com	216
https://chromethemesonline.net	153
https://www.searchcapitol.com/	137
https://pimp-up-your-browser.com	127
https://the-theme-factory.com	124
https://epic-chrome-themes.com	121
https://chrome-themes.online	89
https://tab-e-licious.com	86

Cute HD Wallpaper New Tab[4] is an extension that replaces the `"newtab"` by opening a new website (https://qtab.io/cute81/) and when users introduce a search query in the input bar, the server first redirects the user to another server (inifinitynewtab) that hijack the search query and other information of the user before finally redirecting to Google Chrome. Similarly, Breaking News Tab[5] extension replaces the `"newtab"` web page with another one provided by the extension itself. The difference is that this time the search queries go to another server (searchcapitol.com) that server redirects the user to Bing instead once the query is received.

In Table E.5, we included the Top-10 of most used URLs by extensions to hijack the search queries after performing the discovering strategies previously explained. Finally, we automatically installed all the extensions we found and check whether they indeed open the web pages we found when the user opens a new tab.

As a summary, we analyzed the downloads pattern of browser extensions and detected 1,292 malicious ones that hijack users' searches. We combined this method with Dedup.js to detect similar extensions that were not analyzed because their increment of downloads did not pass the filter (what we call "*sleeping*" extensions) and we found 5,288 extra malicious extensions, totaling 6,579.

E.6 Discussion

Web Store Downloads Granularity In September 2021, Google changed the way the Web Store shows the information affecting the granularity of the number of

[4] ieclinianmfccihifhicbaofnkhndamd
[5] jgginkfhlcakpkjfkkbbcnjpeoladhih

downloads of the extensions. Numbers are no longer as precise, e.g., the same number of downloads of the extension used in Listing E.1 is now represented as "5,000+ users". Although the analysis presented in this paper is based on the precise number of downloads that Google used to offer and we crawled for 6 months, we believe that similar results might still be achievable with the new coarse-grained numbers. More importantly, if our methodology were deployed by vendors, then precise numbers of the downloads would be used. They would also be able to implement our system in real-time and not having to wait for 90 days.

Dynamic analysis While it is unlikely to get false positives using dynamic analysis, we do find cases of false negatives. We identify three main limitations in our dynamic analysis that result in false negatives. These are 1) installation wizards; 2) HTML elements blocking the search bar, and; 3) state destruction.

Some extensions use installation wizards where the user can fill in preferences and import data before being able to use the extensions. Our dynamic analysis makes no attempt at solving these wizards in general. Therefore it can fail to reach the search bar and test it for query stealing. Using our manual and static verification analysis we identified a group of 27 extensions that used `tabturbo.com` for stealing queries. However, due to a multi-step installation wizard, we were not able to reach the search bar.

Another general limitation is invisibility or obstruction of the search bar. For example, `pricehelpers.com` requires the user to click a drop-down menu before being able to search. However, our dynamic approach does not find the drop-down menu needed for the search. The opposite problem is when extensions use modal windows to obstruct the search bar, until the user closes the modal window. In many of the cases, our approach could handle this since many of the modal windows did not cover the entire search bar. In the case of full obstruction, our method would fail. An example of search bar obstruction is Naruto New Tab Page Top Wallpapers Themes[6]. Interestingly, we input DeDup.js with the files of this extension and it detected other 3,637 malicious extensions using mytab.me as an external web server to steal users' queries.

Finally, we use a best-effort approach to find the search bar on the new tab page. If there are multiple text inputs with different functionalities, our approach will only test the first one. We did find one such case where an extension had one search bar for Gmail where it did not steal the query, and a second search bar for Google where it did steal the query. In that particular case, we only checked the first search bar. To improve this, the extension, or at least the new tab page, should be analyzed once per input element.

As with any type of Potentially Unwanted Programs (PUP), the *potentially* part could result in false positives. If users truly want a search engine other than the major ones or if they want their queries to be sent to multiple servers in addition to their normal search engine. To allow for flexibility, we make it easy to change our code, or relabel the data later, to add or remove servers considered safe.

[6] bemmphgeeoaljepcaficneogmlndijbi

Missing Code There were 413 extensions that we were not able to collect the source code before they were removed from the Web Store. In these cases, we followed the presumption of innocence and considered the extensions to be benign. If we could have analyzed their code, our false positive rate would likely have decreased.

Missing Attacks In this paper, we focus on the attack of stealing queries. This means that we might mark an extension as safe even though it performs other malicious actions, like injecting advertisements on pages. As we already find malicious clusters, that even share code-specific parts, as the URLs used, we believe our analysis is strong enough to find malicious clusters. By including more attacks in the security analysis, we might have found more clusters for other attacks that are now marked as safe instead.

Code Similarity While code similarity is useful in many cases, it also has limitations. Indeed, there are cases where our pattern analysis is more effective. One reason is that benign and malicious extensions can have very similar code. For example, the safe extension *jabbaohcijedbmkbdjldjicnohlcdkdp* and the malicious *dbkbnddmcjkjkclnlpagncoebgfaoile* both have a `main.js` file where only 4 out of the 391 lines differ. The main difference here is the URL used by the extension. We can also use techniques based on hash files like DeDup.js [39]. Here, main.js does not match but there are still other JavaScripts, CSSs and images that are exactly the same in both. There have been multiple earlier cases where malicious authors simply copy benign code and change a small part to make them malicious[7]. By focusing on download patterns we are able to distinguish these as two different clusters.

Another reason is that malicious extensions can have a small footprint by relying on external websites, i.e., they do not need dangerous permissions or powerful API calls to steal queries (e.g., [2, 28, 37]). In fact, the previously mentioned malicious extension (dbkbnddmcjkjkclnlpagncoebgfaoile) does not define a single permission. This makes them hard to detect by traditional methods.

E.7 Related Work

Attacks against collaborative filtering recommendation systems have recently attracted much interest to the research community. We can differentiate between white-box [66, 68] and black-box attacks [22, 51, 62]. In white-box attacks, attackers have some knowledge about the recommendation algorithms or how the data (users/items) are related. This knowledge can be gathered by, for instance, interacting with the system similar to what users can do. On the contrary, in black-box attacks, attackers do not know anything about the recommendation system or the implemented algorithms. Examples of real attacks on CFRSs, and more concretely on Android Google Play have been proven to be successful [10, 42, 43, 44] when attackers alter the rating and the number of raters.

[7] `https://freeaddon.com/warning-adware-virus-distributors-are-making-fake-extensions-based-on-freeaddon-sportifytab/`

Recently, Dou *et al.* [18] deployed a honeypot in App Market—an alternative non-official Android store—to track the number of downloads of the apps, categorizing the download fraud problem in mobile App markets. In comparison to our proposal, we proved that the download fraud problem in browser extensions can be directly associated with security attacks and showed how the download pattern can be used to detect malicious extensions.

Browser Extensions In the last years, browser extensions have attracted the interest of the research community. In 2015, Google engineers [28] claimed they catch 70% of the malicious ones within 5 days whereas some extensions can remain months or even years without being detected [5].

A few examples of how private researchers in the *industry* help in detecting malicious extensions are: maladvertising and cryptocurrency [1, 19]; spyware [46]; phishing [40]; proxy scripts [31]; remote code execution [5], and; ransomware [9, 41, 58].

In *academia*, among other attacks, researchers demonstrated that extensions suffer from maladvertising [4, 57, 61, 65]; fingerprinting [32, 33, 48, 49, 52, 53, 54, 55, 59]; JavaScript injection attacks [7, 20, 50], and showed how over-privileged the extensions are [2, 28]. When detecting malicious extensions, researchers usually rely on static [6, 26, 69] and dynamic analysis [11, 30, 36, 55, 70] whereas only a few authors included machine learning techniques [2, 28, 37].

ML in Browser Extensions Jagpal *et al.* [28] were one of the first who use machine learning to detect malicious extensions. In concrete, they used Logistic Regression (LR) to train a model after extracting API, permissions, DOM operations, and behavioral signal of over 90k extensions monitored during 3 years, obtaining an overall recall of 96.5% and a precision of 81% for one year.

Aggarwal *et al.* [2] detect malicious extensions by extracting their API sequence, and using these sequences to train a Recurrent Neural Network (RNN), achieving high precision (90.02%) and recall (93.31%) in detecting spying extensions.

Regarding clustering, recently, Pantelaios *et al.* [37] used DBSCAN to cluster extensions to detect malicious ones based on their reviews, ratings, and descriptions. Despite being a common technique for detecting anomalies in the time-series, i.e., peaks that should not be there, it is not feasible for clustering them.

Contrarily to previous work, we present a novel methodology to automatically detect malicious extensions based on the number of downloads and without having to analyze the source code of the extensions. Similar to previous work, our classifier achieves 0.88 and 0.90 values for precision and recall respectively.

Comparison Compared with Eriksson *et al.* [20], in the over 4,000 extensions they found, none matches ours. This is expected as their method focuses on detecting extensions stealing queries by intercepting traffic from search engines, as opposed to extensions presenting their own search forms.

Compared with the dataset from Pantelaios *et al.* [37], they found 143 extensions. Of these, one appeared in our test set, and we both agreed that it was malicious. In general, their solution requires many reviews on the extensions, something our extensions did not have, making a direct comparison challenging.

Finally, we compared it with DeDup.js, a code similarity method [39]. Here we included JS, HTML and CSS files. Used the malicious extensions in our training set and checked the code similarity with each extension in the test set. Since DeDup.js only calculates the number of files in common, we need to pick a threshold for the number of files needed to be considered "similar". To be as fair as possible, we tried all thresholds between 1 and 50 files, and 1 resulted in the best F-score. The F-score is 0.47 (compared to our 0.89). In more detail, DeDup.js achieves: 349 (TP) 15 (FN) 781 (FP) 67 (TN), whereas our method achieves: 326 (TP) 38 (FN) 44 (FP) 804 (TN). As this shows, DeDup.js finds more malicious extensions than we do but at the cost of a large number of false positives.

E.8 Conclusions

This paper has put the spotlight on the patterns of browser extension downloads and suggested an approach of leveraging these patterns as a signal to drive the analysis of extensions. We collect the download patterns from the Web Store for 6 months to extract our dataset. Using a semi-supervised clustering algorithm, we derive 135 clusters from the 2,858 extensions to discover that the patterns for the extensions in the clusters are often strikingly similar, answering positively our first research question: *Are there extensions that follow similar download patterns?*

We combined static, manual, and dynamic analysis to mark all the 2,858 extensions as malicious or benign, with respect to the attacks in our threat model. Using these security labels with our clusters we showed that 61 of the 135 contain more than 80% malicious extensions. This affirms our second research question: *Is there any relationship between download patterns and malicious code?*

We showed that by creating a classifier trained on download patterns of already deleted extensions we can find the malicious extensions that are still active in the Web Store, affirming our last research question: *Can we find malicious extensions based on their download patterns?*

Finally, driven by the download pattern signal, we leveraged a code-similarity analysis to find a total of 6,579 malicious extensions, uncovering "sleeping" extensions whose download patterns contain too little information.

Bibliography

[1] Over 20,000,000 of chrome users are victims of fake ad blockers. https://adguard.com/en/blog/over-20-000-000-of-chrome-users-are-victims-of-fake-ad-blockers.html, 2021.

[2] A. Aggarwal, B. Viswanath, L. Zhang, S. Kumar, A. Shah, and P. Kumaraguru. I spy with my little eye: Analysis and detection of spying browser extensions. In *Euro S&P*, April 2018.

[3] S. Aghabozorgi, A. S. Shirkhorshidi, and T. Y. Wah. Time-series clustering – a decade review. *Information Systems*, 53:16–38, 2015.

[4] S. Arshad, A. Kharraz, and W. Robertson. Identifying extension-based ad injection via fine-grained web content provenance. In *RAID*, 2016.

[5] Backdoored Browser Extensions Hid Malicious Traffic in Analytics Requests. https://decoded.avast.io/janvojtesek/backdoored-browser-extensions-hid-malicious-traffic-in-analytics-requests/, 2021.

[6] S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, and M. Winslett. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM*, 54(9), 2011.

[7] A. Barua, M. Zulkernine, and K. Weldemariam. Protecting web browser extensions from javascript injection attacks. In *ICECCS*, July 2013.

[8] ios developers use “well-known” download bots to manipulate app store rankings. https://www.cultofmac.com/146438/ios-developers-use-well-known-download-bots-to-manipulate-app-store-rankings-report/.

[9] "Catch-All" chrome extension silently steals your data. https://blog.barkly.com/catch-all-malicious-google-chrome-extension, 2021.

[10] H. Chen, D. He, S. Zhu, and J. Yang. Toward detecting collusive ranking manipulation attackers in mobile app markets. In *Asia CCS*, page 58–70, 2017.

[11] Q. Chen and A. Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *CCS*, 2018.

[12] No more silent extension installs. http://blog.chromium.org, 2021.

[13] T. V. Craenendonck, S. Dumančić, and H. Blockeel. COBRA: A fast and simple method for active clustering with pairwise constraints. In *IJCAI*, pages 2871–2877, 2017.

[14] Cse.google.com - Jan 2021 update. , 2021.

[15] A. Dempster, F. Petitjean, and G. I. Webb. Rocket: exceptionally fast and accurate time series classification using random convolutional kernels. *Data Mining and Knowledge Discovery*, 34(5):1454–1495, 2020.

[16] A. Dempster, D. F. Schmidt, and G. I. Webb. Minirocket: A very fast (almost) deterministic transform for time series classification. In *KDD*, page 248–257, 2021.

[17] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh. Querying and mining of time series data: Experimental comparison of representations and distance measures. In *VLDB*, volume 1, page 1542–1552, Aug. 2008.

[18] Y. Dou, W. Li, Z. Liu, Z. Dong, J. Luo, and S. Y. Philip. Uncovering download fraud activities in mobile app markets. In *ASONAM*, pages 671–678, 2019.

[19] Malicious chrome extensions found in chrome web store, form droidclub botnet. https://blog.trendmicro.com/trendlabs-security-intelligence/malicious-chrome-extensions-found-chrome-web-store-form-droidclub-botnet/, 2022.

[20] B. Eriksson, P. Picazo-Sanchez, and A. Sabelfeld. Hardening the Security Analysis of Browser Extensions. In *SAC*, 2022.

[21] ExtPose - Track your browser extension app store performance and get competitive advantage. , 2022.

[22] W. Fan, T. Derr, X. Zhao, Y. Ma, H. Liu, J. Wang, J. Tang, and Q. Li. Attacking black-box recommendations via copying cross-domain user profiles. In *ICDE*, pages 1583–1594, 2021.

[23] M. Fang, G. Yang, N. Z. Gong, and J. Liu. Poisoning attacks to graph-based recommender systems. In *ACSAC*, page 381–392, 2018.

[24] S. Farooqi, A. Feal, T. Lauinger, D. McCoy, Z. Shafiq, and N. Vallina-Rodriguez. Understanding incentivized mobile app installs on google play store. In *IMC*, page 696–709, 2020.

[25] How are items ranked in the store? `https://developer.chrome.com/docs/webstore/faq/#faq-gen-24`, 2022.

[26] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *S&P*, May 2011.

[27] X. H., Y. Y., L. X., R. L., N. J., and S. W. Time series k-means: A new k-means type smooth subspace clustering for time series data. *Information Sciences*, 367-368:1–13, 2016.

[28] N. Jagpal, E. Dingle, J. Gravel, P. Mavrommatis, N. Provos, M. Rajab, and K. Thomas. Trends and lessons from three years fighting malicious extensions. In *USENIX*, 2015.

[29] A. Kampouraki, G. Manis, and C. Nikou. Heartbeat time series classification with support vector machines. *IEEE Transactions on Information Technology in Biomedicine*, 13(4):512–518, 2009.

[30] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX*, 2014.

[31] Is your browser extension a botnet backdoor. https://krebsonsecurity.com/2021/03/is-your-browser-extension-a-botnet-backdoor/, 2021.

[32] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine. Browser fingerprinting: A survey. *ACM Trans. Web*, 14(2), Apr.

[33] P. Laperdrix, O. Starov, Q. Chen, A. Kapravelos, and N. Nikiforakis. Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets. In *USENIX*, Aug. 2021.

[34] W. Meert, K. Hendrickx, and T. V. Craenendonck. wannesm/dtaidistance v2.0.0, Aug. 2020.

[35] M. O'Mahony, N. Hurley, N. Kushmerick, and G. Silvestre. Collaborative recommendation: A robustness analysis. *ACM Trans. Internet Technol.*, 4(4):344–377, Nov. 2004.

[36] K. Onarlioglu, M. Battal, W. Robertson, and E. Kirda. Securing legacy firefox extensions with sentinel. In *DIMVA*, 2013.

[37] N. Pantelaios, N. Nikiforakis, and A. Kapravelos. You've changed: Detecting malicious browser extensions through their update deltas. In *CCS*, page 477–491, 2020.

[38] J. Paparrizos and L. Gravano. K-shape: Efficient and accurate clustering of time series. In *SIGMOD*, page 1855–1870, 2015.

[39] P. Picazo-Sanchez, M. Algehed, and A. Sabelfeld. DeDup.js: Discovering Malicious and Vulnerable Extensions by Detecting Duplication. In *ICISSP*, 2022.

[40] TA413 Leverages New FriarFox Browser Extension to Target the Gmail Accounts of Global Tibetan Organizations. https://www.proofpoint.com/us/blog/threat-insight/ta413-leverages-new-friarfox-brow ser-extension-target-gmail-accounts-global, 2022.

[41] How dangerous is bad rabbit ransomware and how to avoid it. https://safebytes.com/dangerous-bad-rabbit-ransomware-avoid/, 2021.

[42] M. Rahman, N. Hernandez, B. Carbunar, and D. H. Chau. Search rank fraud de-anonymization in online systems. In *HT*, page 174–182, 2018.

[43] M. Rahman, N. Hernandez, R. Recabarren, S. I. Ahmed, and B. Carbunar. The art and craft of fraudulent app promotion in google play. In *CCS*, page 2437–2454, 2019.

[44] M. Rahman, M. Rahman, B. Carbunar, and D. H. Chau. Fairplay: Fraud and malware detection in google play. In *SDM*, pages 99–107, 2016.

[45] S. Rani, M. Kaur, M. Kumar, V. Ravi, U. Ghosh, and J. R. Mohanty. Detection of shilling attack in recommender system for youtube video statistics using machine learning techniques. *Soft Computing*, pages 1–13, 2021.

[46] Exclusive: Massive spying on users of Google's Chrome shows new security weakness. https://www.reuters.com/article/us-alphabet-google-chrome-exclusive/exclusive-massive-spying-on-users-of-googles-chrome-shows-new-security-weakness-idUSKBN23P0JO, 2021.

[47] I. Sanchez-Rola, M. Dell'Amico, D. Balzarotti, P. Vervier, and L. Bilge. Journey to the center of the cookie ecosystem: Unraveling actors' roles and relationships. In *S&P*, 2021.

[48] I. Sánchez-Rola, I. Santos, and D. Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *USENIX*, 2017.

[49] A. Sjösten, S. V. Acker, P. Picazo-Sanchez, and A. Sabelfeld. Latex gloves: Protecting browser extensions from probing and revelation attacks. In *NDSS*, 2019.

[50] D. F. Somé. Empoweb: Empowering web applications with browser extensions. In *S&P*, May 2019.

[51] J. Song, Z. Li, Z. Hu, Y. Wu, Z. Li, J. Li, and J. Gao. Poisonrec: An adaptive data poisoning framework for attacking black-box recommender systems. In *ICDE*, pages 157–168, 2020.

[52] K. Soroush, I. Panagiotis, S. Konstantinos, and P. Jason. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *NDSS*, 2020.

[53] O. Starov, P. Laperdrix, A. Kapravelos, and N. Nikiforakis. Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *WWW*, 2019.

[54] O. Starov and N. Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *WWW*, 2017.

[55] O. Starov and N. Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *S&P*, 2017.

[56] R. Tavenard, J. Faouzi, G. Vandewiele, F. Divo, G. Androz, C. Holtz, M. Payne, R. Yurchak, M. RuÃŸwurm, K. Kolar, and E. Woods. Tslearn, a machine learning toolkit for time series data. *Journal of Machine Learning Research*, 21(118):1–6, 2020.

[57] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. Mccoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. A. Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *S&P*, May 2015.

[58] Malicious chrome extension steals data posted to any website. https://threatpost.com/malicious-chrome-extension-steals-data-posted-to-any-website/128680/, 2021.

[59] E. Trickel, O. Starov, A. Kapravelos, N. Nikiforakis, and A. Doupé. Everyone is different: Client-side diversification for defending against extension fingerprinting. In *USENIX*, Aug. 2019.

[60] T. Van Craenendonck, W. Meert, S. Dumančić, and H. Blockeel. COBRAS-TS: A new approach to semi-supervised clustering of time series. In *Discovery Science*, pages 179–193. Springer International Publishing, 2018.

[61] G. Varshney, S. Bagade, and S. Sinha. Malicious browser extensions: A growing threat: A case study on google chrome: Ongoing work in progress. In *ICOIN*, Jan 2018.

[62] C. Wu, D. Lian, Y. Ge, Z. Zhu, E. Chen, and S. Yuan. Fight fire with fire: Towards robust recommender systems via adversarial poisoning training. In *SIGIR*, page 1074–1083, 2021.

[63] X. Wu, L. Xiao, Y. Sun, J. Zhang, T. Ma, and L. He. A survey of human-in-the-loop for machine learning, 2021.

[64] X. Xing, W. Meng, D. Doozan, A. C. Snoeren, N. Feamster, and W. Lee. Take this personally: Pollution attacks on personalized services. In *USENIX*, pages 671–686, Aug. 2013.

[65] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee. Understanding malvertising through ad-injecting browser extensions. In *WWW*, 2015.

[66] G. Yang, N. Z. Gong, and Y. Cai. Fake co-visitation injection attacks to recommender systems. In *NDSS*, 2017.

[67] L. Ye and E. Keogh. Time series shapelets: A new primitive for data mining. In *KDD*, page 947–956, 2009.

[68] Y. Zhang, J. Xiao, S. Hao, H. Wang, S. Zhu, and S. Jajodia. Understanding the manipulation on recommender systems through web injection. *IEEE Transactions on Information Forensics and Security*, 15:3807–3818, 2020.

[69] B. Zhao and P. Liu. Behavior decomposition: Aspect-level browser extension clustering and its security implications. In *RAID*, 2013.

[70] R. Zhao, C. Yue, and Q. Yi. Automatic detection of information leakage vulnerabilities in browser extensions. In *WWW*, 2015.

Appendix

E.I Dataset Distribution

In Figure E.11, we extracted the last public download the Web Store offered per extension and represented the distribution of the downloads according to the category they belong to in the Web Store.

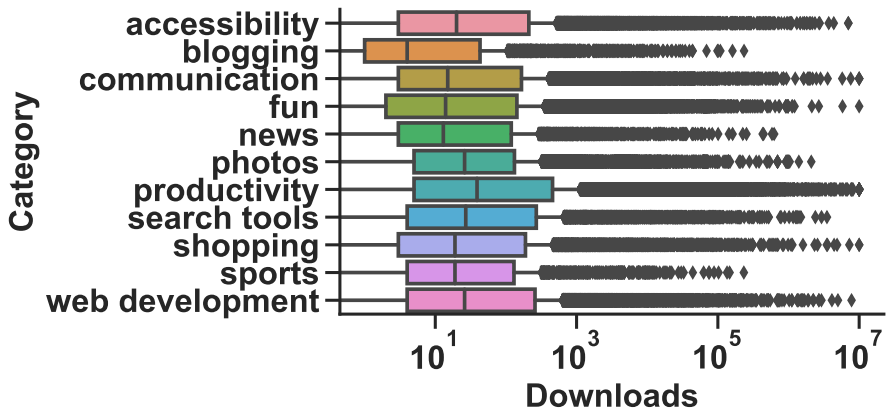


Figure E.11: Distribution of the downloads of the categories the Web Store is composed of.

E.II Source Code

In Listing E.6, we include the manifest file of 29 malicious extensions, having all of them the same manifest but with different icon (<iconFileName>), background page ("js/<name>.js") and version (<version>).

```
{
    "background": {
        "persistent": true,
        "scripts": [
            "js/<name>.js"
        ]
    },
    "browser_action": {},
    "chrome_url_overrides": {
        "newtab": "output.html"
    },
    "default_locale": "en",
    "description": "__MSG_description__",
    "icons": {
        "128": "<iconFileName>"
    },
    "manifest_version": 2,
    "name": "__MSG_name__",
    "offline_enabled": false,
    "update_url": "https://clients2.google.com/service/
    update2/crx",
    "version": "<version>"
}
```

Listing E.6: Manifest file of 1,315 extensions.

E.III Clusters

In Figure E.12 and Figure E.13 we present more of the benign and malicious clusters we find.

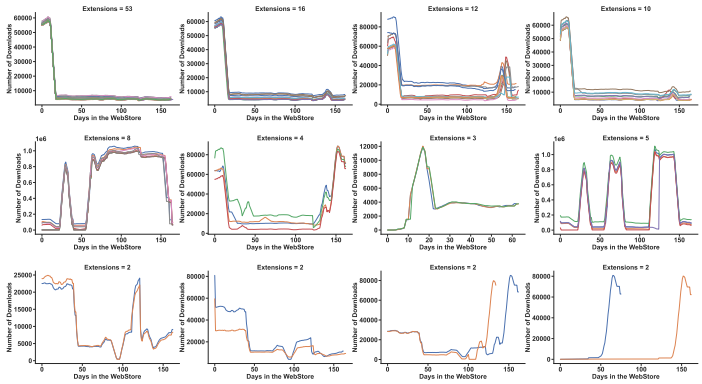


Figure E.12: Benign Clusters.

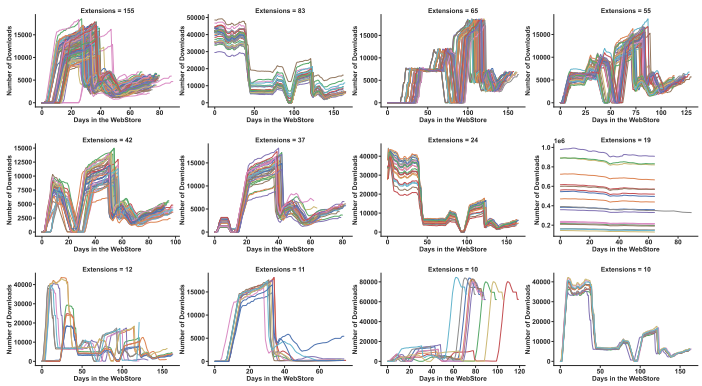


Figure E.13: Malicious Clusters.

Embedded Systems

F

On the Road with Third-Party Apps: Security Analysis of an In-Vehicle App Platform

Abstract. Digitalization has revolutionized the automotive industry. Modern cars are equipped with powerful Internet-connected infotainment systems, comparable to tablets and smartphones. Recently, several car manufacturers have announced the upcoming possibility to install third-party apps onto these infotainment systems. The prospect of running third-party code on a device that is integrated into a safety critical in-vehicle system raises serious concerns for safety, security, and user privacy. This paper investigates these concerns of in-vehicle apps. We focus on apps for the Android Automotive operating system which several car manufacturers have opted to use. While the architecture inherits much from regular Android, we scrutinize the adequateness of its security mechanisms with respect to the in-vehicle setting, particularly affecting road safety and user privacy. We investigate the attack surface and vulnerabilities for third-party in-vehicle apps. We analyze and suggest enhancements to such traditional Android mechanisms as app permissions and API control. Further, we investigate operating system support and how static and dynamic analysis can aid automatic vetting of in-vehicle apps. We develop AutoTame, a tool for vehicle-specific code analysis. We report on a case study of the countermeasures with a Spotify app using emulators and physical test beds from Volvo Cars.

F.1 INTRODUCTION

The modern infotainment system, often consisting of a unit with a touchscreen, is mainly used for helping the driver navigate, listening to music or making phone calls. In addition to this, many users wish to use their favorite smartphone apps in their cars. Thus, several car manufacturers, including Volvo, Renault, Nissan and Mitsubishi [42, 50], have chosen to use a special version of Android for use in cars, called *Android Automotive* [19]. Other manufacturers such as Volkswagen [49] and Mercedes-Benz [31] are instead developing their new in-house infotainment systems. In contrast to the in-house alternatives, Android Automotive is an open platform with available information and code. This justifies our focus on Android Automotive apps.

Android Automotive For the manufacturers, a substantial benefit of using an operating system based on Android is gained from relying on third-party developers to provide in-vehicle apps. A multitude of popular apps already exists on the Android market, which can be naturally converted into Android Automotive apps. Further, Android Automotive is a stand-alone platform that does not require a connected smartphone, contrary to its competitors MirrorLink [34], Apple CarPlay [1] and Android Auto [17].

Safety, security, and privacy challenges While third-party apps boost innovation, they raise serious concerns for safety, security, and user privacy. Indeed, it is of paramount importance that the platform both safely handles these apps while driving and also safeguards the user's privacy-sensitive information against leakage to third parties. Figure F.1 gives a flavor of real-life safety concerns, by showing a user comment on a radio app with almost half a million downloads. The user points out that they had to stop driving when a shockingly loud ad was suddenly played, adding that ads "shouldn't attempt to kill you" [51].

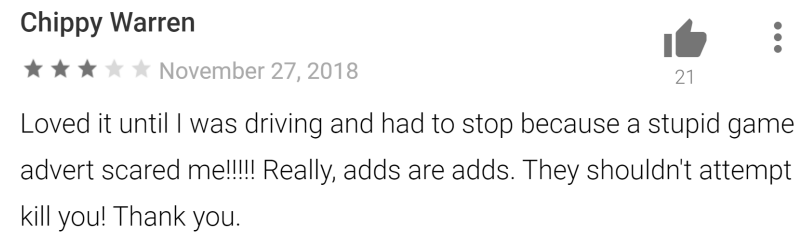


Figure F.1: Top comment on a radio app. A user was shocked by the volume of an ad and had to stop driving.

While the Android Automotive architecture inherits much from regular Android, a key question is whether its security mechanisms are adequate for in-vehicle apps. However, compared to the setting of a smartphone, in-vehicle apps have obvious safety-critical constraints, such as neither being able to tamper with the control system nor being able to distract the driver. Further, car sensors provide sources of private information, such as location and speed or sound from the in-vehicle microphone. In fact, voice controls are encouraged for apps in infotainment systems, as to help keep the driver's hands on the wheel, opening up for audio snooping on users by malicious apps. A recent experiment done by GM collected location data and radio listening habits from its users with the goal of creating targeted radio ads [8]. This clearly highlights the value of user data in vehicles. Similar data could potentially be collected by apps using the radio API to record the current station [15]. Thus, a key question is whether Android's security mechanisms are adequate for in-vehicle apps.

Android Permissions Android's core security mechanism is based on a *permission model* [21]. This model forces apps to request permissions before using the system resources. Sensitive resources such as camera and GPS require the user to explicitly

grant them before the app can use them. In contrast, more benign resources such as using the Internet or NFC can be granted during installation. However, there are several limitations of this model with implications for the in-vehicle setting. From a user's perspective, these permissions are often hard to understand. Porter Felt et al. [40] show that less than a fifth of users pay attention to the permissions when installing an app, and even a smaller fraction understands the implications of granting them. Understanding the implications of giving permissions is even harder. There are immediate privacy risks, such as an app having permission to access the car's position and the Internet can potentially leak location to any third party. More advanced attacks would only need access to the vehicle speed. This may not seem like a privacy issue but by knowing the starting position, likely the user's home address, and speed, it is possible to derive the path that the car drives [16].

Analyzing Android Automotive Security To the best of our knowledge, this is the first paper to analyze application-level security on the Android Automotive infotainment system. To assess the security of the Android Automotive app platform, we need to extend the scope beyond the traditional permissions.

Attack surface For a systematic threat analysis, we need to analyze the attack surface available to third-party apps. This includes analyzing what effects malicious apps may have on the functions of the car, such as climate control or cruise control, and on the driver. We demonstrate *SoundBlast*, representative of disturbance attacks, where a malicious app can shock the driver by excessive sound volume, for example, upon reaching high speed. We also demonstrate availability attacks like *Fork bomb* and *Intent storm* which render the infotainment system unusable until it is rebooted. Further, we explore attacks related to the privacy of sensitive information, such as vehicle location and speed, as well as in-vehicle voice sound. We show how to exfiltrate location and voice sound information to third parties. In order to validate the feasibility of the attacks, we demonstrate the attacks in a simulation environment obtained from Volvo Cars. Based on the attacks, we derive exploitable vulnerabilities and use the Common Vulnerability Scoring System (CVSS) [13] to assess their impact.

To address these vulnerabilities, we suggest countermeasures of permissions, API control, system support, and program analysis.

Permissions We identify several improvements of the permission model. This includes both introducing missing permissions, such as those, pertaining to the location and the sound system in the car, as well as making some permissions more fine-grained. For location, there are ways to bypass the location permission by deriving the location from IP addresses. At the same time, the location permission currently allows all or nothing: either sharing highly accurate position information or not. The former motivates adding missing permissions, while the latter motivates making permissions more fine-grained. We argue that for many apps, like Spotify or weather apps, low-precision in the location, e.g. city-level, suffices.

API control In contrast to permissions, API control can use more information when decided to grant an app access to a resource. For example, using high-precision data could be allowed only once an hour, or during an activity like running. Our findings reveal that apps currently need access to the microphone in order to use

voice controls. We deem this as breaking the principle of least privilege [44]. To address this, we argue for full mediation, so that apps subscribe to voice commands mediated by the operating system, rather than having access to the microphone. Similarly, location data can also be mediated to limit the precision and frequency of location requests, making it possible to adhere to the principle of least privilege. These scenarios exemplify countermeasures we suggest to improve API controls for in-vehicle apps.

System We argue for improvements to the operating system in order to protect against apps using too much of the system's resources. Malicious apps can cause the system to become unresponsive or halt, either by recursively creating new processes or coercing other system processes to use up all the resources. The countermeasures consist of limiting the number of requests an app can make, limiting the resources system processes can use, or completely blocking some capabilities for third-party apps, like creating new processes.

Code analysis While previous methods protect the device from malicious apps, our vision is to also be able to stop the apps before they make it to the device. This can be accomplished by analyzing the code in the app store, before the app is published. This does not only protect against malicious apps but also poorly written apps that fail to adhere to security best practices. This could, for example, include apps not using encryption for data transmissions, which is currently a big problem [41]. Other problems include vulnerable apps with high privileges being exploited by malicious apps or colluding malicious apps sharing data over covert channels. Thus, we investigate how static and dynamic program analysis can be leveraged to address the vulnerabilities.

We design and develop AutoTame, our own static analysis tool for detecting dangerous use of APIs, including the new automotive APIs. AutoTame is open source and will be freely available at the time of publication. Further, we explore several state-of-the-art techniques, based on tools like FlowDroid [3] and We are Family [4].

Threat model The threat model in this paper defines the attacker as being able to install one or more apps, with the victim's permission, on their infotainment system. Similar to previous research [45], we assume that the victim is more inclined to install an app that asks for fewer permissions. This means that, while one app with access to both Internet and GPS might be considered suspicious, two apps, one with access to the Internet, the other with access to GPS, would be more acceptable. Such a model incentivizes apps to collude and share information over covert channels. Using this model we analyze how much damage can be done by a user mistakenly installing malicious apps.

Case study An ideal evaluation of our countermeasures would be a large-scale of apps from an app store, in the style of the studies on Google Play, e.g. [3, 10, 37]. Unfortunately, Android Automotive is at this stage an emerging technology with no apps yet publicly available for a study of this kind. Nevertheless, we have been granted access to Infotainment Head Unit emulators and physical test beds from Volvo Cars allowing us to perform a case study with an in-vehicle app version of Spotify. We use this infrastructure to evaluate our countermeasures.

Impact At the same time, an early study of Android Automotive security has its advantages. Because our analysis comes at an early phase of Android Automotive adoption by car manufacturers, it has higher chances for impact. We have reported our findings to both Volvo Cars that participated in our experiments and Google. We are in contact with both on closing the vulnerabilities we point out and on experimenting with the countermeasures.

Contributions The paper offers the following contributions:

- We present an attack surface for third-party in-vehicle apps, identifying classes of disturbance, availability, and privacy attacks (Section F.3).
- We propose countermeasures, based on fine-grained permissions, API control, system support, and information flow (Section F.4).
- We overview prominent representatives of techniques and tools for detecting security and privacy violations in third-party apps (Section F.4.4).
- We present our own static analysis tool, AutoTame, for detection of dangerous API usage (Section F.4.4).
- We evaluate the countermeasures on a case study with the in-vehicle app Spotify (Section F.5).

F.2 BACKGROUND

As cars become more connected and their infotainment systems more powerful, people expect the car to interact in a seamless way with their other devices. In contrast to most other personal devices, a software bug in a car can have lethal consequences. For example, in 2015 Miller and Valasek [33] showed that it was possible to remotely take over a 2014 Jeep Cherokee by exploiting their infotainment system Uconnect. More recently, in May 2018, researchers found multiple vulnerabilities in the infotainment system and Telematics Control Unit of BMW cars which made it possible to gain control of the CAN buses in the vehicle [48]. These type of attacks show that remote take over attacks of connected vehicles is a possibility and a real threat.

Attackers do not necessarily need to take control over the braking or steering system to endanger or distract the driver. For example, an attacker can make a malicious infotainment app that disturbs or shocks the driver at a certain speed level. In order to shock the driver, the app may, for example, play loud music or rapidly flash the screen.

In addition to security, privacy is also a concern as cars become more capable of collecting data about their users. In accordance with the new EU regulation, GDPR [11], the user has to be informed about how the data is used and agree to their data being used in the described way. Previous research projects have explored the possibility to automatically track and analyze how privacy-sensitive information is leaked from Android apps [43], either deliberately through advertisement networks or inadvertently through insecure communication means [41].

F.2.1 Experimental Setup

With access to Volvo Cars internal testing equipment, both the attacks and countermeasures were tested on their infrastructure. In particular, the code is tested on Volvo's *Infotainment Head Unit emulators (IHU)* emulators and physical test beds. All of the Android code is developed for Android SDK version 26 and 27, which corresponds to Android 8.0 and 8.1.

F.2.2 Automatic analysis of Android apps

Automatically analyzing Android apps can be done through two major strategies, static analysis or dynamic analysis. Static analysis only considers the code while in dynamic analysis the code is executed and the program's behavior is analyzed. Which ever method is chosen, a decision on what to look for in the analysis has to be made. In this paper, two tracks are evaluated, how privacy-sensitive information flows through the app and scanning apps for common vulnerabilities.

F.2.3 Android Automotive

Today, the Android system is officially used in all types of devices, from phones and tablets to watches, TVs and soon cars [18]. Android Automotive is a version of Android developed specifically for use in cars. It is essentially Android with a User Interface (UI) adapted for cars and a number of car specific APIs. The car specific APIs allow for control over vehicle functions, such as the heating, ventilation, and air conditioning (HVAC), and reading of sensor data, e.g. speed, temperature and engine RPM [19]. Android Automotive is not be confused with Android Auto which is already available on the market today. Unlike Android Auto, Automotive is a completely stand-alone system that is not dependent on a smartphone. In Android Auto, apps run on the users Android phone which then renders content on a screen in the car. The apps and the Android system thus runs separated from the car.

F.2.4 Android's Permission model

The Android operating system controls access to many parts of the system, such as camera, position and text messages, through permissions. These permissions can be of one of four types; *normal*, *dangerous*, *signature* or *signatureOrSystem*. The first two are the most common and can be granted to any third-party app. Normal permissions give isolated accesses with minimal risk for the system and user, these are automatically granted by the operating system. Dangerous permissions, on the other hand, give accesses to private user data and control over the device that may harm the user. These permissions have to be explicitly granted by the user on a per application basis. Both Android's *coarse* and *fine* location permissions are examples of dangerous permissions, since both supply high precision data. The difference between them is that *fine* location has access to the GPS while *coarse* uses cell towers and WiFi access points. Finally, there are the *signature* and *signatureOrSystem* permissions, which requires the app to be pre-installed or cryptographically signed [20].

Table F.1: The attacks are divided into three different categories. Which asset and permissions the attacks affects and requires are listed along with the needed user interaction.

Name	Category	Asset	User interaction	Permission	Severity
SoundBlast	Disturbance	Driver's attention	Start app	None	Medium[a]
Fork bomb	DoS	CPU resources	Start app	None	Medium
Intent storm	DoS	CPU resources	Start app	None	Medium
Permissionless speed	Privacy	Current speed	Start app	None	Low
Permissionless exfiltration	Privacy	Data Exfiltration	Start app	None	Low
Covert channel	Privacy	Data Exfiltration	Start app	Channel dependent	Low

[a] The score is subject to the limitation of CVSS3 on lacking support for physical damage and safety risks [9].

F.2.5 Covert channels

A *covert channel*, as defined by Lampson, is a communication channel between two entities that are not intended for information transfer [25]. In Android, a number of different covert channels exist that use both hardware attributes and software functions to communicate. Apps can for example communicate by reading and setting the volume, sending special intents or cause high and low system load [29, 45].

F.3 ATTACKS

This section focuses on the implementation decisions regarding the attacks presented in Table F.1. The category and asset columns in the table give an understanding of what the attack is targeting. More specifically, the asset is what the attack is trying to take control over. In the case of denial-of-service (DoS) attacks, this is usually some type of resource. Privacy attacks, on the other hand, try to acquire and exfiltrate data such as speed or location. User interaction and permission are used to judge how easy the attack is to execute. The values are finally combined to create a severity score based on the Common Vulnerability Scoring System (CVSS3) [13]. A shortcoming of CVSS3 is that possible physical damage or safety risks are not considered in the scoring. Distraction vulnerabilities, like the one exploited by SoundBlast, and other automotive vulnerabilities will be underrated. These shortcomings are currently being revised for CVSS3.1 [9]. The exact vectors and scores for each attack are presented in Table F.3. Table F.2 present the same attacks together with suitable countermeasures to mitigate the underlying vulnerabilities.

F.3.1 Disturbance

SoundBlast The SoundBlast attack relies heavily on the `AudioManager` class in Android. This class supplies functions which are used to control the volume of different audio streams in Android. Cars also have the more specific `CarAudioManager`, however, this class requires special permissions. Different audio streams are used to differentiate between volumes, e.g. music volume, ringer volume, alarm volume, etc. A malicious app can use the permissonless audio API to max the volume and shock the driver. The attack is further improved by using a `ContentObserver` to listen for changes in volume and force the volume to the maximum as soon as it

changes. Using the vehicle's sensors, the attacker can also design the attack to only active when traveling at high speeds.

Testing the SoundBlast attack shows that it is possible to set any volume on all the different audio streams in Android, without needing any permissions. In addition, the attack can also detect changes in volume and max the volume accordingly. The changes are also detected even if the driver uses the hardware controls on the IHU or steering wheel. Killing the app is the only way to regain control of the volume.

F.3.2 Availability

Fork bomb A fork bomb is a program that creates new instances of itself until the system runs out of resources, either freezing the device or force a reboot. While this might be acceptable on a phone, in a vehicle setting this is problematic. Since the IHU usually handles navigation, freezing the device might distract drivers trying to fix it, or frustrate them by having to stop and reboot.

Forking in Android is not possible by default, resulting in the need for a vulnerability to leverage in order to accomplish forking. Unlike previously successful fork bomb attacks on Android [2], our attack takes an application-level approach by creating a shell, which in turn has the power to fork itself. Similar to other programming languages, Android also supports a version of `exec`, which can be used to run external programs. However, this is not enough to create a new process that can copy itself. By using `exec` to run `sh -s`, a new shell is created, which in turn can execute the fork bomb.

When testing this attack it is able to fully grind both the emulator and test bed to a halt, requiring a power cycle to regain control. It is thus able to render the infotainment system unusable until the system is rebooted.

Intent storm The intent storm attack uses Android intents to continuously restart the app itself. Similar to the fork bomb presented in section F.3.2, the intent storm attack tries to use up all the CPU resources, making the IHU unusable. The difference, however, is that the intent storm does not use the resources itself, but rather forces another system process, the *system_server*, to use up all resources. The fast activity switching required is made possible with threads and intents. As soon as the app starts, it spins up 8 threads which all ask Android to start its own main activity. Using multiple threads increases the pressure on the *system_server*, making the device less responsive.

During the tests, the *system_server* process was forced by the attack to use 100% of the CPU, making the IHU unusable. In some cases, an error message popped up on the device prompting the user to either kill or wait for the app. Regardless of which alternative was picked, the attack would continue without interruption since a request to restart the app had already been sent. Similar to the fork bomb in section F.3.2, this would grind the IHU to a halt. However, in some cases, the IHU would automatically restart after a few minutes.

F.3.3 Privacy

Permissionless speed In Android Automotive, apps have direct access to the current speed. However, since speed is privacy sensitive it requires a permission. By combining other permissionless sensor values, such as the current RPM and gear, and knowledge about the wheel size, the speed can be derived. The effectiveness of this attack does depend on the sampling frequency of the sensors. The hardware test beds only contained the IHU and not the full car, meaning that the efficiency of the attack is yet to be tested.

Permissionless exfiltraion The Android permission model clearly states that any app wanting communicate on a network requires Internet permission. However, by using intents it is possible to force another app with Internet permission to leak the data. Depending on how the intent is crafted, different apps will handle them, for example, the web browser will open URLs, music player opens music files, etc.

While the implementation details differ depending on which app handles the intent, the common procedure is to encode the data, split it into chunks and send a separate intent for each chunk.

While the default web browser can be used, there are better options for exfiltrating data. By changing the data type to `audio/wav` and using the URL `http://evil.com/music.wav?d=[data]`, the music player will load the URL instead. The stealthiness of this method depends on which music player is used. Using the native Android music player, a small popup with a play button will appear. By returning a malformed wav file from the server, the music player will show a more subtle error message.

If a web browser is used, the attacker can have the server redirect the request to a deep link, giving control back to the exfiltration app. Not only does this give the app the ability to leak more data, but it also enables two-way communication with the attacker's server, all without using the Internet permission.

In order to test this, a proof-of-concept code was developed that would record audio for five seconds and then upload it using the described method. The code only needs permission to record audio, but not to use the Internet. Testing this attack shows that it is possible to send data to the Internet without using the Internet permission. The attack was successful using Chrome, the standard music player, video player and image viewer. If the device has not been configured with a default application for opening the type of data, it will ask the user to pick one.

Covert channels Previous work on covert channels in Android have used both vibration and volume settings to transmit data between colluding apps [45]. While these are still viable in Android Automotive, there are also additional new interesting APIs. In particular the new climate control API for temperature. Since the temperature is represented by a floating point value, the bandwidth is more than tenfold that of the volume settings. However, changing the temperature does currently require a signature permission, making it hard for third-party apps to acquire.

In contrast to previous work on covert channels, which relied on time synchronization, our attack is based on asynchronous messages. This forces the receiver to send an acknowledgment for each of the received values. While this lowers the bit

rate, in contrast to synchronous communication, it greatly increases the reliability of the communication.

With this implementation, two apps can collude to leak privacy- sensitive information to the Internet. One app requests permission to privacy-sensitive information but not the Internet and then acts as a sender. The second app requests Internet permission but not permission to access any sensitive data. The second app can now receive sensitive information which it does not have permission for and leak it to the Internet.

F.4 COUNTERMEASURES

The vulnerabilities are very different in nature and, as such, the mitigation techniques differ. Some vulnerabilities can be mitigated by several different techniques while others can only be mitigated by one. An overview of the attacks together with mitigations for the underlying vulnerabilities are presented in Table F.2.

F.4.1 Permission

The current permission model can be improved both by adding new permissions for unprotected resources, and also by refining some very broad permission. The SoundBlast attack, from Section F.3.1, relies on changing the volume through an API called *AudioManager* which does not require any sort of permission. At the same time, there exists an API called *CarAudioManager*, which does require a permission. Cars usually have more advanced sound systems than phones so a different API with more settings does make sense as does the need for a permission. Still, when conducting experiments with the emulator the *AudioManager* is present and usable by third-party apps, thus allowing an attacker to circumvent the permission required by *CarAudioManager*.

In addition to audio, Android allows apps to get the location of the device by using GPS. This can, for example, be used by apps to give weather information. However, due to these systems having high precision and allowing for multiple requests within short time intervals, apps often excessive information.

There are multiple methods for preserving the user's privacy while still maintaining an acceptable level of functionality in apps using location [12, 32]. Which method is optimal is highly dependent on the type of information the app needs. A simple approach is to truncate location, effectively creating a grid of possible locations. A grid will better protect the privacy of the user, but at the same time degrade the functionality of some apps [32]. In order to handle apps like fitness trackers, which requires fast updates and high precision, truncation is not feasible. Fawaz and Shin [12] argue that in order to preserve privacy, a choice has to be made between tracking distance and speed, or tracking the path of the exercise. They present a method for tracking the distance and speed by supplying the exercise tracker with a synthetic route, that has correct distance and speed but a forged path. Furthermore, they argue that navigation apps with Internet access, usually used for real-time traffic information, are the hardest to handle since they can potentially leak the location.

This problem could be solved by using state-of- the-art information flow tracking to ensure that the location is never leaked.

F.4.2 API control

In some scenarios, permissions are not enough. This is usually the case when access to a resource can be abused over time. For example, in the current Android model, apps are allowed to record audio from the microphone at all times, as long as it has been granted the permission once. This means that a restaurant app that uses voice commands to find close by restaurants, can listen to everything the user says, at all times. Since voice commands are more prevalent in vehicles, where the user's focus is on driving, it is reasonable to believe that more in-vehicle apps will use this functionality. One solution to this problem is to use a voice mediator, which is a special service that has access to the microphone and allows for third-party apps to subscribe to certain keywords. The app would only receive sentences that contain the keywords it subscribed to, effectively removing its capabilities to eavesdrop. Similar to the voice mediation, the same method can be used for location. By using a location mediator apps can subscribe to arbitrary precision for location data. The mediator can also introduce a trade-off between the refresh rate and precision of the requests, mitigating real-time tracking.

F.4.3 System

Some problems are best solved at the operating system level. These problems include resource management, e.g. how much CPU time or memory an app should be allowed to use. One method of limiting the impact of availability attacks is by limiting how frequent a resource can be acquired. Android already does this to a great extent when it comes to memory and CPU usage by third-party apps. However, some system processes, the *system_server* process in particular, can use all of the CPU, effectively starving the rest of the system. This lack of rate limiting was exploited in the intent storm attack in Section F.3.2. While not tested, we speculate that this vulnerability could either be countered by rate limiting the CPU usage of the *system_server* process or limit incoming intents to the *system_server*.

Similar to CPU limiting, memory usage requires limitations too. When Android is running low on memory it will start to terminate apps in the background. This can sometimes result in the termination of apps that the user wants to run in the background. In the case of vehicles, navigation apps are a good example of apps that should not be killed of while driving. A possible method for ensuring that the navigation works while driving is to prohibit Android from terminating important apps. This protects against both malicious apps using up the memory, and legitimate memory hungry apps.

Akin to permissions, SELinux policies are policies which limit what the processes in an OS can do. These policies play a crucial role in protecting the vehicle's subsystems from Android. The policies are also suitable for specifying what an app is allowed to do. However, not how many times it can do it. As Bratus et al. [5]

explains, "SELinux does not provide an easy way to control the use of the fork operation once forking has been allowed in the program's profile", which shows that SELinux is not suited to stop attacks like fork bombing. While it might be infeaseable in many situations, blocking forking altogether could be a solution.

F.4.4 Code analysis

Automatic analysis techniques can be used to scan apps, both before installation and during runtime, to find vulnerabilities and block attacks. In the following sections tools using these techniques are described in more detail.

Vulnerability detection Both AndroBugs [26] and QARK [27] are tools that can be used to scan Android apps for known vulnerabilities. QARK is capable of finding many common security vulnerabilities in Android apps [22]. QARK can, for example, find incorrect usage of cryptographic functions, trace intents and detect insecure broadcasts. In addition, QARK can also generate exploits for some of these vulnerabilities. While not able to generate exploits, AndroBugs can detect vulnerabilities based on heuristics in the code. For example, multiple dex files suggests a master key vulnerability (CVE-2013-4787) [35]. The tools work well together since AndroBugs can quickly scan multiple apps with heuristics and then QARK can perform a deeper analysis of the interesting apps.

AutoTame To scan for dangerous use of the new automotive APIs, we developed a special tool built on the Soot framework, which can analyze both Java and Android bytecode. The tool has a list of dangerous APIs, e.g controlling the HVAC system, change audio volume or spawning shells. Using Soot, our tool decompiles the APK and analyses each function in the app while testing if it matches any of the ones in the list. AutoTame performs a full application analysis. The main advantage of this is that it does not require any entry point analysis. Compared to many other languages, Android apps do not have a single main function from which execution starts. Therefore a full analysis ensures that any dangerous use of an API is detected. However, without knowing the entry points, dead code could be flagged, potentially leading to false positives. In addition to only detecting if the volume is changed, AutoTame can also give extra warnings if the volume is set to a high numeric value or if `getStreamMaxVolume` is used. If a match is found the app can be removed or marked as potentially dangerous. The tool was able to flag the SoundBlast attack, as well as the fork bomb.

Taint tracking Taint tracking can help detect privacy leaks where sensitive information, such as the user's location, is being sent to a remote server. FlowDroid [3] is a tool for static taint analysis on Android, that can detect these flows. The taint analysis works by tainting private sources of information, such as the user's location. If the location is written to a variable, then this variable also becomes tainted. If at a later time this tainted variable is written to a public sink, e.g an Internet connection, a leak from a private source to a public sink will be detected.

What makes FlowDroid special is its highly accurate modeling of Android's life cycles. This is important as an app can be started in many different ways. In addition

Table F.2: List of all developed attacks and which countermeasure(s) can be used to mitigate each attack the underlying vulnerabilities.

Attacks / Countermeasures	Permissions	Location granularity	SELinux	AutoTame	FlowDroid	We are Family	Rate limit
SoundBlast	✓			✓			
Fork bomb			✓	✓			
Intent Storm							✓
Permissionless speed		✓			✓	✓	
Permissionless exfiltration		✓			✓	✓	
Covert channels	✓	✓			✓	✓	

to life cycles, FlowDroid is also able to track callback functions, enabling it to track leaks via button clicks and other UI events. Important for the car API used in this paper is that FlowDroid can track dynamically registered callback functions, which is used to establish the connection to the car.

In order to make FlowDroid fully functional with Android Automotive apps, we extended the tool with new sources and sinks. Some of the sources added were used to acquire the car's manufacturer, model and year. For sinks, we added functions for writing to the climate control APIs.

Observable flows Taint tracking is not always enough to find all privacy leaks. For this reason, a more powerful tool that can detect observable implicit flows is introduced. The *We are Family* paper by Balliu et al. [4] presents a two-fold hybrid analysis solution. The first stage is a static analysis that transforms the application and adds monitors. These monitors will aid the dynamic analysis tool in the second stage to find implicit flows. The added monitors are in this case used to track the program counter label and analyze the current taint value, making it possible to detect potential leaks during runtime on the device. The dynamic tool developed in the paper is an extension of TaintDroid [10]. By using the transformed program together with TaintDroid, the new tool is able to detect observable implicit flows, something TaintDroid was not able to do.

F.5 SPOTIFY CASE STUDY

To test some of the countermeasures, an in-depth case study was performed on the Spotify app. The motivation behind using Spotify is that it was the only third-party app available on the emulator and test bed, making it the most realistic app to test. It was also much larger in size than the proof-of-concept attacks. The larger size will show how well the methods handle real apps.

F.5.1 Permissions

The first analysis that has to be performed is to gather an understanding of the permissions the app uses. Spotify needs permission to Internet, Bluetooth and NFC, for data transfer. Furthermore, it also requires permission to change audio settings, run at startup, and prevent the device from sleeping. Since Spotify is a music streaming app that should be able to run in the background, as well as talk to other Bluetooth devices, these permissions seem innocuous. Shifting focus to the *dangerous* permissions, Spotify does require permission to read the accounts on the device, contacts

stored on the device, the device ID, and information about current calls. It is not clearly motivated why this information is necessary, and while some connection between the Spotify user and the device user is reasonable, having access to all contacts seems excessive. Spotify does not ask for the location permission, instead, they use IP-addresses for location [47]. In addition, Spotify can also record audio and take pictures, as well as read and write access to the external storage. Taking pictures is necessary to scan QR-codes and the microphone will be used in Spotify's driving mode [46]. Access to external storage is reasonable since it allows for offline storage of music, however, it does include access to other photos and media files beyond Spotify's.

F.5.2 Vulnerability detection

To ensure that the app does not have any known vulnerabilities QARK is used to scan the app. While QARK didn't find any severe vulnerabilities, it did find cases where a vulnerability could arise, e.g. by using a WebView in an older version of Android (API $\leq$ 18). Moreover, it also points out interesting entry-points into the app, one of them leading to a version of Spotify meant for another automotive system. In addition, a malicious third-party app can also send intents to Spotify to search and play arbitrary music, skip songs, or even crash the app. QARK did not find any vulnerabilities relating to the vehicle APIs, motivating the need for further analysis.

F.5.3 AutoTame

Using AutoTame, multiple warnings about both changing the volume and querying for max volume was found. Further manual analysis proved that the maximum volume was used directly to set the volume, as shown in Figure F.2.

```
1  int i = this.c.getStreamMaxVolume(0);
2  this.c.setStreamVolume(0, i, 0);
3
```

Figure F.2: Decompiled code setting volume to max

F.5.4 Information flow analysis

The permissions give an upper bound on what the app is capable of doing. A more precise understanding of the app is achieved by analyzing it with FlowDroid, using implicit flow tracking. Using these settings the information flow analysis found 13 leaks in the app. One interesting leak was `getLastKnownLocation` being leaked into a dynamic receiver registration. As shown in Figure F.3, FlowDroid was able to track the sensitive location through different assignments, function calls and control flows. While this case might be quite benign, as it only leaks one bit, it still shows the capabilities of the technique.

The analysis also over-approximates some leaks, especially when the information being sent is based on information being received. A concrete example of this is when threads try to communication using `sendMessage` and `obtainMessage`. Since the obtained information could contain sensitive information, it is flagged as a leak. This could potentially be solved using dynamic information flow tracking.

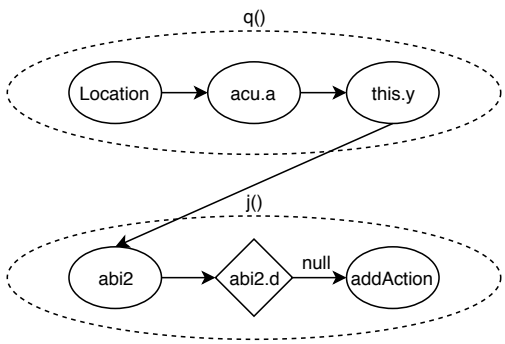


Figure F.3: The publicly observable addAction function is implicitly dependent on the private location information.

F.5.5 Summary

To summarize these findings, we see that a more robust and at the same time more fine-grained permission model would be beneficial, as it would allow apps like Spotify to use lower precision location data instead of privacy-invading high precision data. In addition, vulnerability detection methods succeed in finding a bug that could be exploited to terminate Spotify. Finally, static analysis proved successful for automatically detecting privacy leaks.

F.6 RELATED WORK

Previous security and privacy research on vehicles have to a large extent focused on low-level problems relating to the internal components. Koscher et al. [24] showed that with physical access to the CAN bus it is possible to control both the speedometer, horn and in-vehicle displays to distract the driver. Miller and Valasek [33] gained similar access to the CAN bus, this time remotely. A similar vulnerability found in an infotainment system used in cars from Volkswagen was also recently discovered by researchers in the Netherlands [7]. They showed that it was possible to connect to the car via WiFi to exploit a service running in the infotainment system to gain remote code execution system. The most recent study on attacks against vehicles were done by researchers at Tencent Keen Security Lab [48], where they found multiple vulnerabilities in the infotainment system and Telematics Control Unit of BMW cars, resulting in control of the CAN buses.

A contribution of our paper is to show that even without access to the internal buses or exploiting low-level vulnerabilities, it is possible to cause distractions and leak private information.

A more high-level study was done by Mazloom et al. [30] where they conducted a security analysis of the MirrorLink protocol. MirrorLink allows smartphones to run apps on the cars infotainment system. Their analysis showed weaknesses in the MirrorLink protocol which could, amongst other things, allow malicious smartphone apps to play unwanted music or interfere with navigation. Mandal et al. [28] showed that the similar system Android Auto have multiple problems that can be abused by third-party apps. For example, auto playing audio when launching an app or showing visual advertisements, both which are against Android Auto's quality policy. In our paper, we show similar attacks are possible on Android Automotive, however, without the requirement of the user's smartphone, since the malicious app runs on the infotainment system.

Intents, which is the main component in our exfiltration attack, are problematic for many reasons. Khadiranaikar et al. [23] highlighted some of these problems, including how malicious apps can both steal information and compromise other apps using intents. Our paper builds on these ideas to develop new exfiltration methods for the Android Automotive platform.

There is a large body of work on Android permissions [14, 38, 39]. As a representative example, a study on Android permissions by Porter Felt et al. [37] shows that many apps are using more permissions that they need, i.e. not adhering to the principle of least privilege. Other researches [6], also argue for the need of a more fine-grained model which can grant access to specific functions instead of full APIs or services. Extensions such as Apex [36] have also been developed in order to supply end users with a more fine-grained model, capable of granting permissions based on user-specified policies. While our focus is on the specifics of the in-vehicle setting, we argue that many apps get access to more data than necessary due to the coarse granularity of the permission model itself. For example, a weather app or Spotify app only needs low-precision location, such as city level.

F.7 CONCLUSIONS

To the best of our knowledge, we have presented the first study to analyze application-level security on the Android Automotive infotainment system. Unfortunately, our analysis shows that in-vehicle Android apps are currently as secure as regular phone apps. We argue it is insufficient because in-vehicle apps can affect road safety and to some extent user privacy.

Our study of the attack surface available to third-party apps include driver disturbance, availability, and privacy attacks, for which there is currently no protection mechanisms in Android Automotive.

Consequently, it is important for car manufacturers that third-party apps are limited in their abilities to cause a considerable distraction for the driver. Additionally, there are a number of vehicle specific APIs, such as access to current gear and engine RPM, that is a cause for concern when it comes to user privacy.

To address the vulnerabilities that lead to these attacks, we have suggested the countermeasures of robust and fine-grained permissions, API control, system support, and program analysis.

We have designed and developed AutoTame, a tool for detecting dangerous vehicle-specific API usage. We have demonstrated that in-vehicle code analysis can be performed using AndroBugs and QARK, to detect known vulnerabilities, AutoTame to detect vehicle specific vulnerabilities and FlowDroid, with the additional vehicle specific sources and sinks, to detect privacy leaking apps.

We have evaluated the countermeasures with a Spotify app using an infrastructure of Volvo Cars.

Bibliography

[1] Apple. Apple carplay, 2014. http://www.apple.com/ios/carplay/.

[2] A. Armando, A. Merlo, M. Migliardi, and L. Verderame. Would you mind forking this process? a denial of service attack on android (and some countermeasures). In *IFIP*, 2012.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[4] M. Balliu, D. Schoepe, and A. Sabelfeld. We Are Family: Relating Information-Flow Trackers. In *European Symposium on Research in Computer Security*, 2017.

[5] S. Bratus, M. E. Locasto, B. Otto, R. Shapiro, S. W. Smith, and G. Weaver. Beyond selinux: the case for behavior-based policy and trust languages. 2011.

[6] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *USENIX Security Symposium*, 2013.

[7] Computest. Research paper: The connected car - ways to get unauthorized access and potential implications. Technical report, April 2018.

[8] Detroit Free Press. Gm tracked radio listening habits for 3 months: Here's why, 2018. https://eu.freep.com/story/money/cars/general-motors/2018/10/01/gm-radio-listening-habits-advertising/1424294002/.

[9] D. Dugal. List of potential improvements for cvss 3.1, 2018.

[10] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.

[11] European Commission. Regulation (eu) 2016/679, 2016. http://ec.europa.eu/justice/data-protection/reform/files/regulation_oj_en.pdf.

[12] K. Fawaz and K. G. Shin. Location privacy protection for smartphone users. In *SIGSAC '14*, 2014.

[13] FIRST.Org Inc. Common vulnerability scoring system v3.0: User guide, 2018.

[14] M. Frank, B. Dong, Porter Felt, and D. Song. Mining permission request patterns from android and facebook applications. In *ICDM '12*. IEEE, 2012.

[15] A. Gampe. Radiotestfragment, 2018. https://android.googlesource.com/platform/packages/services/Car/+/4d1e3469cb2f285e7a4a864bd48a4c5177e7c83f/tests/EmbeddedKitchenSinkApp/src/com/google/android/car/kitchensink/radio/RadioTestFragment.java.

[16] X. Gao, B. Firner, S. Sugrim, V. Kaiser-Pendergrast, Y. Yang, and J. Lindqvist. Elastic pathing: Your speed is enough to track you. In *ubicomp 2014*, 2014.

[17] Google Inc. Android auto, 2014. https://www.android.com/auto/.

[18] Google Inc. Android, 2018. https://www.android.com/.

[19] Google Inc. Automotive, 2018. https://source.android.com/devices/automotive/.

[20] Google Inc. permission, 2018. https://developer.android.com/guide/topics/manifest/permission-element.html.

[21] Google Inc. Permissions overview, 2018. https://developer.android.com/guide/topics/permissions/overview.

[22] F. Ibrar, H. Saleem, S. Castle, and M. Z. Malik. A study of static analysis tools to detect vulnerabilities of branchless banking applications in developing countries. In *ICTD '17*, 2017.

[23] B. Khadiranaikar, P. Zavarsky, and Y. Malik. Improving android application security for intent based attacks. In *IEMCON 2017*, Oct 2017.

[24] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, 5 2010.

[25] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, Oct. 1973.

[26] Y.-C. Lin. Androbugs framework, 2018. https://github.com/AndroBugs/AndroBugs_Framework.

[27] LinkedIn Corporation. Qark, 2018. https://github.com/linkedin/qark.

[28] A. K. Mandal, A. Cortesi, P. Ferrara, F. Panarotto, and F. Spoto. Vulnerability analysis of android auto infotainment apps. In *CF '18*. ACM, 2018.

[29] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *ACSAC '12*, 2012.

[30] S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy. A security analysis of an in-vehicle infotainment and app platform. In *WOOT*, 2016.

[31] Mercedes-Benz. Mercedes-benz user experience: Revolution in the cockpit, 2018. https://www.mercedes-benz.com/en/mercedes-benz/innovation/mbux-mercedes-benz-user-experience-revolution-in-the-cockpit/.

[32] K. Micinski, P. Phelps, and J. S. Foster. An empirical study of location truncation on android. *Weather*, 2:21, 2013.

[33] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015, 2015.

[34] MirrorLink. Mirrorlink, 2009. https://mirrorlink.com/.

[35] MITRE. CVE-2013-4787. Available from MITRE, CVE-ID CVE-2013-4787., 2013. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4787.

[36] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS '10*, 2010.

[37] Porter Felt, E. Chin, S. Hanna, D. Song, and D. A. Wagner. Android permissions demystified. In *ACM Conference on Computer and Communications Security*, pages 627–638. ACM, 2011.

[38] Porter Felt, S. Egelman, M. Finifter, D. Akhawe, D. Wagner, et al. How to ask for permission. In *HotSec*, 2012.

[39] Porter Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission redelegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[40] A. Porter Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. A. Wagner. Android permissions: user attention, comprehension, and behavior. In *SOUPS*, page 3. ACM, 2012.

[41] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Studying tls usage in android apps. In *CoNEXT '17*, 2017.

[42] Renault–Nissan Alliance. Renault-nissan-mitsubishi and google join forces on next-generation infotainment, 2018. https://www.alliance-2022.com/news/renault-nissan-mitsubishi-and-google-join-forces-on-next-generation-infotainment/.

[43] I. Reyes, P. Wiesekera, A. Razaghpanah, J. Reardon, N. Vallina-Rodriguez, S. Egelman, and C. Kreibich. " is our children's apps learning?" automatically detecting coppa violations. In *ConPro'17*, 2017.

[44] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1975.

[45] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS '11*, 2011.

[46] M. Singleton. Spotify is testing a driving mode feature, 2018. https://www.theverge.com/2017/7/7/15937284/spotify-driving-mode-feature-testing.

[47] Spotify. Privacy policy, 2018. https://www.spotify.com/us/legal/privacy-policy/.

[48] Tencent Keen Security Lab. New vehicle security research by keenlab: Experimental security assessment of bmw cars, 2018. https://keenlab.tencent.com/en/2018/05/22/New-CarHacking-Research-by-KeenLab-Experimental-Security-Assessment-of-BMW-Cars/.

[49] Volkswagen. 2018 passat press kit, 2018. https://media.vw.com/en-us/press-kits/2018-passat-press-kit.

[50] Volvo Car Group. Volvo cars to embed google assistant, google play store and google maps in next-generation infotainment system, 2018. https://www.media.volvocars.com/global/en-gb/media/pressreleases/228639/volvo-cars-to-embed-google-assistant-google-play-store-and-google-maps-in-next-generation-infotainme.

[51] C. Warren. Radio fm, 2018. https://play.google.com/store/apps/details?id=com.radio.fmradio&hl=en&reviewId=gp%3AAOqpTOFWacIVZQ-JHULA86lKu5ZYSNQdIjsM8e6Ph0aj2RWN2aVmoFJFfmJhC91yQEErw6Z0Re3I0LF6k1V_o_Y.

Appendix

F.I Attacks and severity score

Table F.3: List of attacks and their severity score, based on CVSS v3.

Name	CVSS v3 Vector	Score
SoundBlast	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:L/A:L	4.4
Fork bomb	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H	5.9
Intent storm	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H	5.9
Permissionless speed	AV:L/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N	3.3
Permissionless exfiltration	AV:L/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N	3.3
Covert channel	AV:L/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N	3.3