ENHANCING THE SECURITY AND PRIVACY OF FULL-STACK JAVASCRIPT WEB APPLICATIONS

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

DISSERTATION

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
von

CRISTIAN-ALEXANDRU STAICU, M.SC.

geboren in Orăștie, Rumänien.

Referenten:

Prof. Dr. Guido Salvaneschi
Prof. Dr. Michael Pradel
Prof. Dr. Andrei Sabelfeld

Tag der Einreichung: 05.02.2020
Tag der Prüfung: 18.03.2020

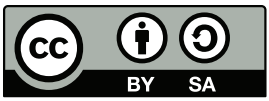
Cristian-Alexandru Staicu: *Enhancing the Security and Privacy of Full-Stack JavaScript Web Applications* , January 2020.

This document was published using tuprints, the E-Publishing-Service of TU Darmstadt.

http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Please cite this document as:

URN: urn:nbn:de:tuda-tuprints-118087
URL: https://tuprints.ulb.tu-darmstadt.de/id/eprint/11808



ERKLÄRUNG

Hiermit erkläre ich, dass ich die vorliegende Arbeit – abgesehen von den in ihr ausdrücklich genannten Hilfen – selbständig verfasst habe.

Darmstadt, Deutschland, März 2020

Cristian-Alexandru Staicu

ACADEMIC CV

October 2014 - March 2020
Doctoral Degree in Computer Science
Technische Universität Darmstadt, Germany

September 2012 - August 2014
Master Degree in Computer Science
University of Trento, Italy and University of Twente, Netherlands

September 2007 - August 2011
Bachelor Degree in Computers and Information Technology
Politehnica University Timișoara, Romania

ABSTRACT

Web applications are the most important gateway to the Internet. Billions of users are relying on them every day and trusting them with their most sensitive data. Therefore, ensuring the security and privacy of web applications is of paramount importance. Traditionally, the server-side code of websites was written in languages such as PHP or Java for which security issues are well studied and understood. Recently, however, *full-stack JavaScript web applications* emerged, which have both their client-side and server-side code written in this language.

We hypothesize that there are several unique properties of full-stack JavaScript web applications that pose a serious challenge for the security analysts: the new threat model for JavaScript, the excessive code reuse, the prevalence of code transformations, and the existence of complex full-stack threats. In this dissertation, we support this thesis by performing several in-depth studies of the JavaScript ecosystem and by proposing multiple improvements to the state-of-the art practices. First, we discuss two types of security vulnerabilities that are aggravated by the new threat model: injections and regular-expression denial of service. Second, we show that excessive code reuse in the JavaScript ecosystem increases the chance of relying on malicious or vulnerable code. Third, we provide evidence that code transformations are widespread and that full-stack threats exist. Finally, we propose several improvements for techniques aimed at hardening web applications: cost-effective consideration of implicit flows, the extraction of taint specification for third-party libraries, and pragmatic program analysis for defending against injections.

The problem of securing full-stack JavaScript web applications is far from settled, but we hope that the current dissertation serves as motivation for future work to consider this increasingly important class of applications. In particular, we argue for holistic approaches that consider full-stack and cross-library information flows.

ZUSAMMENFASSUNG

Webanwendungen sind die wichtigste Schnittstelle zum Internet. Milliarden Nutzer sind täglich auf sie angewiesen und vertrauen ihnen ihre sensiblen Daten an. Deshalb ist es besonders wichtig, die Sicherheit und den Datenschutz von Webanwendungen zu gewährleisten. Der serverseitige Code von Websites wurde üblicherweise in Sprachen wie PHP oder Java geschrieben, deren Sicherheitslücken gut erforscht und nachvollziehbar sind. In letzter Zeit sind jedoch Full-Stack-JavaScript-Webanwendungen aufgetreten, deren clientseitiger und serverseitiger Code in dieser Sprache geschrieben ist.

Unsere Hypothese ist, dass viele Eigenschaften von Full-Stack-JavaScript-Webanwendungen eine ernsthafte Herausforderung für einen Sicherheitsanalysten darstellen: Das neue Bedrohungsmodell für JavaScript, die übermäßige Wiederverwendung von Code, verbreitete Code-Transformationen und komplexe Full-Stack-Bedrohungen. In dieser Dissertation vertreten wir diese These, indem wir das JavaScript-Ökosystem mehrmals gründlich untersuchen und zahlreiche Verbesserungen zum heutigen Stand der Technik aufzeigen. Als Erstes erörtern wir zwei Arten von Sicherheitslücken, die durch das neue Bedrohungsmodell verschärft werden: Injections und Denial-of-Service für reguläre Ausdrücke. Als Zweites wird aufgezeigt, dass eine übermäßige Wiederverwendung von Code im JavaScript-Ökosystem die Wahrscheinlichkeit erhöht, sich auf bösartigen oder anfälligen Code zu verlassen. Als Drittes weisen wir die weite Verbreitung von Code-Transformationen und das Vorkommen von Full-Stack-Bedrohungen nach. Schließlich werden verschiedene verbesserte Techniken zum Härten von Webanwendungen vorgestellt: Berücksichtigung der Effizienz von impliziten Datenflüssen, Extraktion von Taint-Spezifikationen für Bibliotheken von Drittanbietern und pragmatische Programmanalyse zur Abwehr von Injection-Angriffen.

Das Sicherheitsproblem von Full-Stack-JavaScript-Webanwendungen ist noch lange nicht gelöst. Jedoch hoffen wir, dass die vorliegende Dissertation dazu motiviert, diese zunehmend wichtige Applikationsart für zukünftige Arbeiten zu berücksichtigen. Insbesondere befürworten wir ganzheitliche Ansätze, die Full-Stack und bibliotheksübergreifende Informationsflüsse miteinschließen.

ACKNOWLEDGEMENTS

The work described in this dissertation was done over the course of five crazy, intense years. This wonderful time of my life was filled with a lot of transformative experiences. Even though pursuing a PhD is viewed as a way to climb into or build one's ivory tower, I found that the opposite was true in my case: I met a lot of wonderful, down-to-earth people on the way who supported and encouraged my efforts.

First and foremost, I want to thank my adviser Michael Pradel who was an excellent mentor and role model. He guided every step of my PhD adventure and enabled me to transform from a software engineer with an interest for science into an independent researcher. This rite of passage involved climbing many steep slopes and fighting a lot of my inner beasts. Thank you, Michael, I am forever grateful for your immense trust and support. Please do not forget that `if (secret) x = true`.

Next, I want to thank the current and past members of Software Lab. The friendly atmosphere in the group and the accent on high quality research allowed me to continuously grow, while nurturing the right amount of skepticism. In particular, I am very grateful to Andrea Püchner, Marija Selakovic, Marina Billes, Jibesh Patra, Andrew Habib and Daniel Lehmann for their continuous support, for helping with navigating the German bureaucracy, and for the fun philosophical discussions. I also want to thank Markus Zimmermann, Philippe Skolka and Patrick Mell for completing their theses under my supervision and Katharina Plett for helping with the translation of the thesis abstract into German.

I am also grateful for having had the opportunity to co-author papers with great researchers from around the world: Max Schäfer, Anders Møller, Martin Toldam Torp, Nikos Vasilakis, Daniel Schoepe, Musard Balliu, Andrei Sabelfeld, Benjamin Livshits, Luca Della Toffola, Cam Tenny, Hui Liu, Qiurong Liu, Yue Luo, Esben Andreasen, Liang Gong, Koushik Sen, Mariano Ceccato, Paolo Falcarin, Alessandro Cabutto and Yosief Weldezghi Frezghi. Thank you for guiding my research in the right direction in these early years, I learned a lot by working side-by-side with you.

As I mentioned earlier, I had the benefit of meeting a lot of wonderful people while in Darmstadt, many of whom I already mentioned. My colleagues at Software Lab and their partners were some of my closest friends during these years. I want to thank especially Supriti Sinhama-

hapatra, Jovan Krunić and Lydia Gad for the many great afternoons we spent together. Additionally, I want to thank Nikolay Matyunin for his great sense of humor and for our endless discussions about the Eastern European experience and Wen Wang for her extraordinarily positive attitude. Karina Köhres for helping me better understand and fit in the German culture, and the Darmstädter one in particular. Hanne Weismann and Matthew Geddes for our fantastic board game nights, especially for our trip to Chamstone. I also want to thank Peter Merz and Nathalie Brunner Merz for their affection, Giorgia Azzurra Marson for being an amazing flatmate, Tommaso Gagliardoni for his entertaining travel stories and Patrick Struck for the fun pub quiz evenings. Finally, I also want to thank Nikolaos Athanasios Anagnostopoulos, Carel van Rooyen and Spyros Boukoros for tackling together the shock of our first doctorate year.

Next, I would like to thank my teachers and mentors who, during my studies, instilled respect and curiosity for science in me, and for knowledge in general: Marius Minea, Radu Marinescu, Emilia Petrișor, Massimiliano Sala, Artur Kuczapski, Călin Don, Gabriel Petric, Letiția Rafiliu, Maria Pascu and Camelia Tălmaciu.

During the years, I was privileged enough to always be surrounded by like-minded people, each of them leaving a significant mark on my personality. It is impossible for me to thank them all individually here, but I will highlight some important groups. I want to thank my colleagues from the EIT Digital master school for our great time in Trento, Enschede and Eindhoven. Also, I am grateful to *Corabia Nebunilor* and *Poiana lu' Occam*, my groups of friends from Timișoara for sharpening my debating skills at our endless discussions in Retro. I also want to thank my colleagues from my time at industry, especially the ones at Semmle, Philips and acp-IT.

I want to thank my parents, Călin and Cornelia, whose unconditional love and support were the main enablers for all my later achievements: *Sunteți cei mai minunați părinți!* Special thanks also to my uncle Băroi Cornel and to my sister Denisa-Mădălina Lupu for always being there for me. Next, I want to thank the rest of my family, both in Romania and in Hungary for supporting me all these years: *Mulțumesc!* and *Köszönöm!*

Finally, I want to thank my most important supporters. First, my wonderful wife, Ágnes, for always believing in me, for supporting my every step and for making me a better person. Last, but not least, my baby daughter, Anna, for putting a smile on my face every single morning, for letting me rest just the right amount of time while writing this dissertation and, thus, for giving me the opportunity to become a morning person.

CONTENTS

1

INTRODUCTION

It is estimated that almost half of the global population accesses the Internet, with figures as high as 70% in the developed countries. *Web applications* are the main vehicle for surfing the Internet, with some reporting billions of regular active users[1]. Many of these web applications center their business model around user data, which is their main asset. Naturally, in recent years, such a valuable resource became the prime target for attackers. Large multinational organizations reported data breaches affecting millions or sometimes billions of users.

The largest data breach in history was reported by Yahoo! and occurred in 2013 and 2014 when possibly all of its three billion user accounts were compromised. First, the attacker deployed a *targeted attack* called spear phishing that lured employees of the companies into clicking a dangerous link. This link deployed a malware, which further allowed access to the internal network. Using this backdoor, the attacker obtained access to hashed user passwords which were then used for faking web cookies, and thus breaking into user accounts. According to the FBI[2], the adversary exported the database containing private user information, i.e. names, phone numbers and hashed passwords, and made it readily accessible on his server in order to monetize it on the black market. This data was further used for breaking into individual user accounts.

Another infamous example is the Equifax data breach from 2017 when sensitive information about almost half of the US population was illegally accessed. The attackers exploited a known *vulnerability in a software library* (CVE-2017-5638) that the web application developers failed to patch. This attack vector is so popular that the OWASP foundation included a new entry in their latest top 10 most critical security risks for web applications: "A9 - using components with known vulnerabilities".

As illustrated by the previously discussed security incidents, attackers exploit different parts of a web application when mounting their attacks,

1 https://www.statista.com/statistics/432390/active-gmail-users/, https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/

2 https://www.fbi.gov/wanted/cyber/alexsey-belan

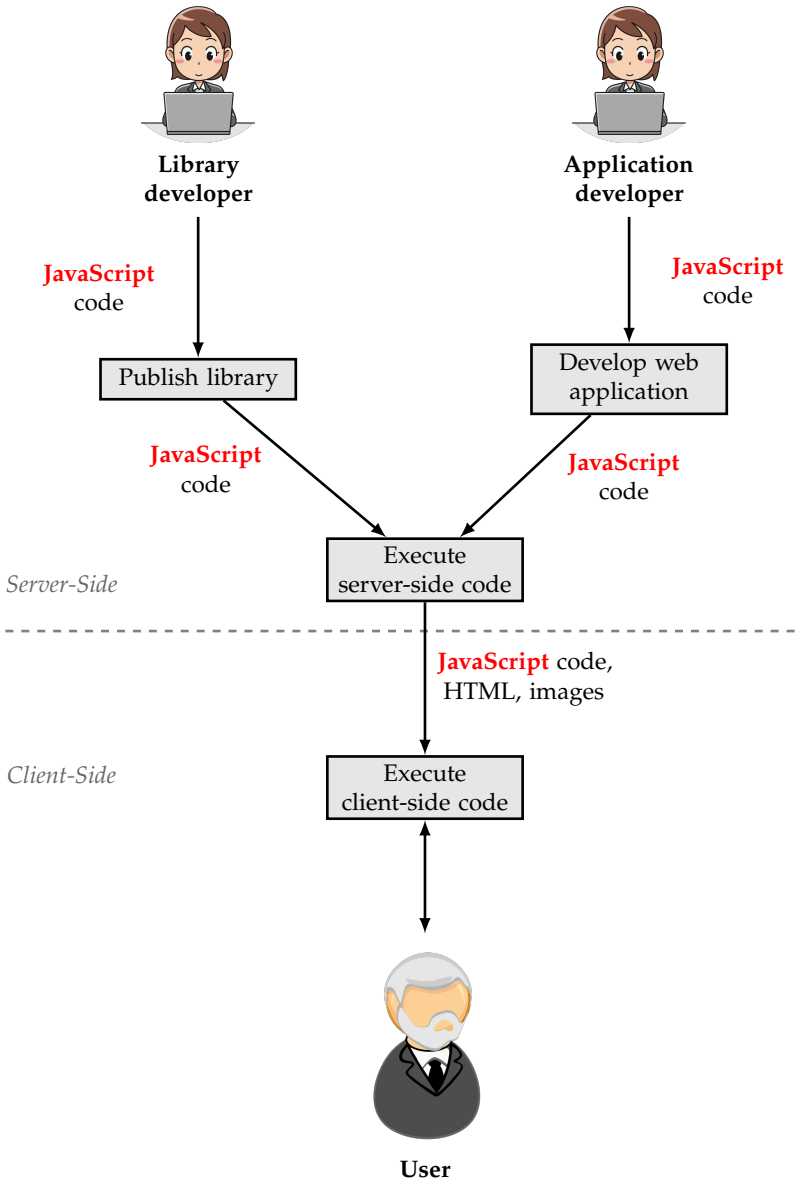


FIGURE 1.1: High-level overview of interactions in a full-stack JavaScript application. On each edge, the code can be transformed using code transformation tools.

e.g., vulnerable software components or web cookies. Therefore, when securing web applications one should consider all their composing parts, ideally in a holistic way. In Figure 1.1 we show a high-level overview of a modern full-stack JavaScript web application. Traditionally, JavaScript was only executed in the browser, but in recent years it became a popular server-side language. We say that an application is full-stack JavaScript if both its client-side and its server-side code is written in this programming language. The advantages of doing so are multi-fold: easier knowledge transfer, uniform usage of tools, and code reuse.

One may hope that security tools and practices traditionally used for client-side code or for other server-side programming languages suffice for hardening this new class of applications. However, there are several reasons why that is not the case and why these applications need to be treated separately by the security community. Below, we discuss several particularities of full-stack JavaScript web applications:

P_1: NEW THREAT MODEL FOR JAVASCRIPT CODE As mentioned earlier, JavaScript traditionally runs inside a browser where the access to sensitive resources is mediated through a security sandbox. On the server-side there is no analogous mechanism, and all the executed JavaScript code, including third-party libraries, has access to the entire available API, e.g., spawn a new process, modify files on the disk, and open ports. To put it differently, there is no default privilege separation or code isolation mechanism in the current server-side JavaScript platforms. This can lead to serious security incidents, such as complete server takeover.

P_2: EXCESSIVE CODE REUSE The largest JavaScript repository for server-side libraries, npm, has an unusually high average number of direct and indirect dependencies per library [DMC16]. This is caused by the thin API of the language and by the excessive usage of trivial [Abd+17] or micro packages [Kul+17]. What is also remarkable about this ecosystem, when compared with similar ones for other languages, is the high number of security incidents that affected it in the recent years. At first, the fragility of the ecosystem caused serious availability issues for numerous libraries, i.e., until recently, developers could delete at will their published code, impacting in real time all their transitive dependencies. Even though this problem was addressed, the lack of automatic code vetting leaves the door open to malware attacks: an adversary can compromise a given popular library and release a new version containing a malicious payload. Excessive

code reuse directly increases the probability of relying on vulnerable or malicious components, a devastating problem as illustrated by the Equifax incident discussed earlier.

P_3: CODE TRANSFORMATIONS JavaScript code is rarely shipped in its original form. Very often code transformation tools are applied either for compacting the code size or for preventing reverse engineering, e.g., for protecting intellectual property. This can impact the activities of a security analyst in multiple ways. First, because in the transformation process certain elements are either removed from the source code, e.g., comments or semantically-rich identifier names, or added, e.g., dead code or unnecessary function calls, the performance of certain code analysis techniques can be degraded by the transformation tools. Second, since JavaScript code is shipped to the client-side, the transformation itself can serve as a side-channel revealing the tools used on the server-side. Once an attacker has such information, she can try to influence the development process of these tools, e.g., by including backdoors.

P_4: FULL-STACK THREATS Existing automated techniques for securing web applications are limited to either client-side or server-side code. However, certain threats can only be detected if both sides of the application are analyzed by an end-to-end tool. This is increasingly the case due to the tendency to push more computation on the client-side and due to the increasingly more powerful Web APIs, e.g., WebRTC, Push API, or WebSockets. Full-stack JavaScript web application are not unique in this regard, but the fact that they use a single language across the stack present a unique opportunity for future work to propose full-stack security analysis tools.

Considering all these particularities, this dissertation supports the following thesis:

> *Full-stack JavaScript web applications present unique challenges and opportunities to the security analysts that need to be addressed by novel tools and practices.*

We support this claim (i) by presenting new attacks enabled by the emerging server-side threat model for JavaScript, (ii) by introducing a novel targeted client-side attack that would require full-stack program analysis for automatic detection, (iii) by showing evidence that code transformation techniques are widely used in web applications and (iv) by dis-

cussing in detail the impact of excessive code reuse and ways for remedying it.

While we emphasize the need for holistic practices that consider security problems end-to-end, across the stack, we advise the reader that the current dissertation puts a special emphasis on securing third-party libraries. Such components are deployed practically in every part of a web application. Nevertheless, we provide evidence that an attacker can exploit vulnerabilities in such libraries to build attacks against live, full-stack JavaScript applications. Moreover, by automatically analyzing the semantics of these libraries we show that one can improve the performance of existing security analyses.

1.1 OUTLINE OF THE THESIS

This dissertation consists of three parts: (i) state of the ecosystem, (ii) vulnerabilities and attacks, and (iii) defenses. Every part consists of individual chapters, each of them supporting the thesis statement by discussing at least one of the aforementioned particularities of full-stack JavaScript web applications. For each chapter, we denote the corresponding particularity described above in brackets.

1.1.1 *State of the Ecosystem*

In Chapter 2 we present an empirical study [Zim+19] of the npm ecosystem, the largest repository of JavaScript libraries in the world. We show how excessive code reuse in the ecosystem (P_2) can lead to potentially devastating malware attacks on the ecosystem. To quantify this problem, we show that an average library depends on 79 third-party libraries and on code managed by 39 maintainers. Moreover, we show that vulnerabilities are a problem as well, transitively affecting up to 40% of the ecosystem. Finally, we discuss a series of ways to improve the state of the npm ecosystem.

In Chapter 3, we propose training an unsupervised machine learning model for distinguishing between transformed and non-transformed code (P_3) and for identifying the tool that was used in the transformation process [SSP19]. We show that this approach is effective at identifying transformed code, closely matching the expectation of expert users. We then use this model in an empirical study of thousands of live websites. Our study shows that minification is widespread, that more complex obfusca-

tion is rare yet non-negligible, and that particular obfuscation techniques are clearly dominant.

1.1.2 *Vulnerabilities and Attacks*

We discuss in detail two classes of vulnerabilities that are amplified by the difference in threat model between client- and server-side (P_1): injection vulnerabilities [SPL18] in Chapter 4 and regular expression denial of service (ReDoS) [SP18] in Chapter 5. We provide evidence that these problems are widespread in server-side JavaScript libraries and that developers are slow to address them. Moreover, in Chapter 5 we discuss a methodology that allows an attacker to leverage vulnerabilities in publicly available libraries for attacking live websites.

In Chapter 6 we discuss leaky images [SP19], a novel privacy attack that exploits exceptions in the same origin policy for targeted deanonymization of users of popular web applications across origins. We also present different flavors of this attack: a group and a scriptless variant. We show that multiple high-profile websites are vulnerable to this attack and we convince several of them to fix the problem. Finally, we discuss that automatically deciding if a website is vulnerable to this attack or not requires complex full-stack reasoning (P_4).

1.1.3 *Defenses*

Our defenses are tailored for finding vulnerabilities in third-party server-side libraries (P_1,P_2), but as we discuss in Chapter 5, these components have direct impact on the security of full-stack JavaScript applications.

In Chapter 7 we propose Synode [SPL18], a lightweight static analysis for identifying possible injection vulnerabilities coupled with a runtime enforcement mechanism. We show that Synode is effective, efficient, and has few false positive. Static analysis is a good fit for this problem because the injection vulnerabilities tend to be locally contained. However, applying such a solution to security problems with non-local information flows can be very challenging.

To address these limitations of static analysis, we explore the possibility of using full-fledged information flow control. Therefore, in Chapter 8 we propose iFlow [Sta+19], a dynamic program analysis that allows the user to customize which type of information flows to be considered: explicit or different types of implicit. In an empirical study with several real-world

vulnerabilities and exploits we show that tracking only explicit flows, i.e., taint analysis, is enough for detecting integrity problems in non-malicious server-side JavaScript code.

In Chapter 9 we advocate a hybrid solution that combines the best of both worlds: we dynamically obtain precise information about the highly-used parts of certain libraries, and use this information in a static analysis to analyze, at scale, clients of these libraries. We propose Taser, a specification extraction mechanism that has at its core a dynamic taint analysis. We show that Taser can extract useful specifications for popular JavaScript libraries that can subsequently be used to improve the effectiveness of a commercial, static program analysis.

Finally, Chapter 10 discusses related work and Chapter 11 concludes by highlighting future research directions for improving the security and privacy of full-stack JavaScript web applications.

1.2 Contributions

As further discussed in Section 1.3, the current dissertation is based upon peer-reviewed pieces of work, each containing several contributions, validated by the research community. At a high level, though, we identify a set of directions or research themes:

server-side fingerprinting attacks We show that an attacker can obtain important information about the libraries and tools used on the sever-side solely by interacting with the live system, i.e., without having access to the server-side code. First, using ReDoS vulnerabilities in open-source libraries, she can find out whether these libraries are used or not. Second, using machine learning models, an attacker can identify transformation tools, i.e., minification and obfuscation, used by the server-side developers. Moreover, by using authenticated image requests, she can deanonymize users of the website.

cost-effective vulnerability detection We present several program analyses for detecting vulnerabilities in JavaScript code: static (Synode), dynamic (iFlow) and hybrid (Taser). The unifying theme for all these tools is the pragmatic design decision to favor scalability and performance over completeness and soundness guarantees. For instance, when building Synode we decide to only perform a lightweight intra-procedural analysis due to the particularities of the vulnerability class we consider and ignore complicated

inter-procedural cases. Similarly, when building Taser, we only consider explicit flows due to the limited value provided by implicit flows in vulnerability detection. Naturally, these program analyses can easily be bypassed by a motivated adversary who is aware of these design decisions. However, we believe that in the absence of such adversaries, i.e., when we are interested in errors made by otherwise trustworthy developers, such pragmatic tools suffice.

COMMUNITY ACKNOWLEDGED VULNERABILITIES Several security advisories were created based on the research work presented in this dissertation, and we were awarded multiple bug bounties, showing that the security problems we describe are relevant to practitioners. Moreover, we are the first to show the link between exploiting vulnerabilities in JavaScript libraries and attacking live websites.

1.3 LIST OF PUBLICATIONS AND OPEN-SOURCE IMPLEMENTATIONS

This dissertation is based on several peer-reviewed publications from which it verbatim reuses material:

1. [Sta+20] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. *Extracting Taint Specifications for JavaScript Libraries*, International Conference on Software Engineering (ICSE), 2020,

2. [Sta+19] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld, *An Empirical Study of Information Flows in Real-World JavaScript*, Workshop on Programming Languages and Analysis for Security (PLAS), 2019,

3. [SP19] Cristian-Alexandru Staicu and Michael Pradel, *Leaky Images: Targeted Privacy Attacks in the Web*, USENIX Security Symposium, 2019,

4. [Zim+19] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel, *Small World with High Risks: A Study of Security Threats in the npm Ecosystem*, USENIX Security Symposium, 2019,

5. [SSP19] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel, *Anything to Hide? Studying Minified and Obfuscated Code in the Web*, The Web Conference (WWW), 2019,

6. [SP18] Cristian-Alexandru Staicu and Michael Pradel, *Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers*, USENIX Security Symposium, 2018,

7. [SPL18] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits, *SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS*, Network and Distributed System Security Symposium (NDSS), 2018.

In Table 1.1 we show the mapping between these publications and different chapters in this dissertation.

USENIX Security Symposium 2019 [Zim+19]	Chapter 2
The Web Conference 2019 [SSP19]	Chapter 3
The Network and Distributed System Security Symposium 2018 [SPL18]	Chapter 4, Chapter 7
USENIX Security Symposium 2018 [SP18]	Chapter 5
USENIX Security Symposium 2019 [SP19]	Chapter 6
The Workshop on Programming Languages and Analysis for Security 2019 [Sta+19]	Chapter 8
International Conference on Software Engineering 2020 [Sta+20]	Chapter 9

TABLE 1.1: Mapping between peer-reviewed publications and different chapters.

In order to encourage future work to reuse our results, in Table 1.2, we compile a list of links to research artifacts, i.e., experimental results or tools. Moreover, we present a list of publicly disclosed vulnerabilities[3] uncovered by our research work. This list consists of tens of CVEs acknowledged by the community, the majority of which were evaluated as medium to high severity.

3 http://software-lab.org/projects/cris_vulnerabilities.html

Chapter 2	https://github.com/markuszm/npm-analysis
Chapter 3	http://software-lab.org/projects/obfuscation_study.html
Chapter 5	https://github.com/sola-da/ReDoS-vulnerabilities
Chapter 7	https://github.com/sola-da/Synode
Chapter 8	https://new-iflow.herokuapp.com/download-iflow.html
Chapter 9	http://brics.dk/taser/

TABLE 1.2: Mapping between chapters and research artifacts.

Part I

State of the Ecosystem

2

SECURITY THREATS IN THE NPM ECOSYSTEM

In this chapter we empirically study different attack scenarios against the largest JavaScript software repository in the world. We consider several threat models that correspond both to software components going rogue, i.e., malware, and to software components containing errors introduced by developers, i.e., vulnerabilities. Our main contribution is to study how heavy code reuse (see particularity P_1 in the introduction) increases the likelihood of certain attacks to occur and to show that this problem is getting worse as time passes. While the scope of this chapter is limited to analyzing standalone software components rather than complete applications, such third-party components are widely used both on the client-side and on the server-side of web applications. Moreover, in Chapter 5 we show how an attacker can use vulnerabilities in third-party code to attack live, full-stack JavaScript websites. This chapter shares material with the corresponding publication [Zim+19].

2.1 MOTIVATION

The *node package manager*, or short *npm*, provides hundreds of thousands of free and reusable code packages to support JavaScript developers with third-party code. The npm platform consists of an online database for searching packages suitable for given tasks and a package manager, which resolves and automatically installs dependencies. Since its inception in 2010, npm has steadily grown into a collection of over 800,000 packages, as of February 2019, and will likely grow beyond this number. As the primary source of third-party JavaScript packages for the client-side, server-side, and other platforms, npm is the centerpiece of a large and important software ecosystem.

The npm ecosystem is open by design, allowing arbitrary users to freely share and reuse code. Reusing a package is as simple as invoking a single command, which will download and install the package and all its transitive dependencies. Sharing a package with the community is similarly easy, making code available to all others without any restrictions or checks. The openness of npm has enabled its growth, providing packages for any

situation imaginable, ranging from small utility packages to complex web server frameworks and user interface libraries.

Perhaps unsurprisingly, npm's openness comes with security risks, as evidenced by several recent incidents that broke or attacked software running on millions of computers. In March 2016, the removal of a small utility package called *left-pad* caused a large percentage of all packages to become unavailable because they directly or indirectly depended on left-pad.[1] In July 2018, compromising the credentials of the maintainer of the popular *eslint-scope* package enabled an attacker to release a malicious version of the package, which tried to send local files to a remote server.[2]

Are these incidents unfortunate individual cases or first evidence of a more general problem? Given the popularity of npm, better understanding its weak points is an important step toward securing this software ecosystem. In this chapter, we systematically study security risks in the npm ecosystem by analyzing package dependencies, maintainers of packages, and publicly reported security issues. In particular, we study the potential of individual packages and maintainers to impact the security of large parts of the ecosystem, as well as the ability of the ecosystem to handle security issues. Our analysis is based on a set of metrics defined on the package dependency graph and its evolution over time. Overall, our study involves 5,386,239 versions of packages, 199,327 maintainers, and 609 publicly known security issues.

The overall finding is that the densely connected nature of the npm ecosystem introduces several weak spots. Specifically, our results include:

- Installing an average npm package introduces an implicit trust on 79 third-party packages and 39 maintainers, creating a surprisingly large attack surface.
- Highly popular packages directly or indirectly influence many other packages (often more than 100,000) and are thus potential targets for injecting malware.
- Some maintainers have an impact on hundreds of thousands of packages. As a result, a very small number of compromised maintainer accounts suffices to inject malware into the majority of all packages.

1 https://www.infoworld.com/article/3047177/javascript/how-one-yanked-javascript-package-wreaked-havoc.html
2 https://github.com/eslint/eslint-scope/issues/39

- The influence of individual packages and maintainers has been continuously growing over the past few years, aggravating the risk of malware injection attacks.
- A significant percentage (up to 40%) of all packages depend on code with at least one publicly known vulnerability.

Overall, these findings are a call-to-arms for mitigating security risks on the npm ecosystem. As a first step, we discuss several mitigation strategies and analyze their potential effectiveness. One strategy would be a vetting process that yields trusted maintainers. We show that about 140 of such maintainers (out of a total of more than 150,000) could halve the risk imposed by compromised maintainers. Another strategy we discuss is to vet the code of new releases of certain packages. We show that this strategy reduces the security risk slightly slower than trusting the involved maintainers, but it still scales reasonably well, i.e., trusting the top 300 packages reduces the risk by half. If a given package passes the vetting process for maintainers and code, we say it has "perfect first-party security". If all its transitive dependencies pass the vetting processes we say that it has "perfect third-party security". If both conditions are met, we consider it a "fully secured package". While achieving this property for all the packages in the ecosystem is infeasible, packages that are very often downloaded or that have several dependents should aim to achieve it.

2.2 Security Risks in the npm Ecosystem

To set the stage for our study, we describe some security-relevant particularities of the npm ecosystem and introduce several threat models.

2.2.1 *Particularities of Npm*

Locked dependencies In npm, dependencies are declared in a configuration file called package.json, which specifies the name of the dependent package and a version constraint. The version constraint either gives a specific version, i.e., the dependency is *locked*, or specifies a range of compatible versions, e.g., newer than version X. Each time an npm package is installed, all its dependencies are resolved to a specific version, which is automatically downloaded and installed.

Therefore, the same package installed on two different machines or at two different times may download different versions of a dependency. To

solve this problem, npm introduced package-lock.json, which developers can use to lock their transitive dependencies to a specific version until a new lock file is generated. That is, each package in the dependency tree is locked to a specific version. In this way, users ensure uniform installation of their packages and coarse grained update of their dependencies. However, a major shortcoming of this approach is that if a vulnerability is fixed for a given dependency, the patched version is not installed until the package-lock.json file is regenerated. In other words, developers have a choice between uniform distribution of their code and up-to-date dependencies. Often they choose the later, which leads to a technical lag [DMC18] between the latest available version of a package and the one used by dependents.

HEAVY REUSE Recent work [DMC17; Kik+17] provides preliminary evidence that code reuse in npm differs significantly from other ecosystems. One of the main characteristic of the npm ecosystem is the high number of transitive dependencies. For example, when using the core of the popular Spring web framework in Java, a developer transitively depends on ten other packages. In contrast, the Express.js web framework transitively depends on 47 other packages.

MICROPACKAGES Related to the reuse culture, another interesting characteristic of npm is the heavy reliance on packages that consist of only few lines of source code, which we call *micropackages*. Related work documents this trend and warns about its dangers [Abd+17; Kul+17]. These packages are an important part of the ecosystem, yet they increase the surface for certain attacks as much as functionality heavy packages. This excessive fragmentation of the npm codebase can thus lead to very high number of dependencies.

NO PRIVILEGE SEPARATION In contrast to, e.g., the Java security model in which a SecurityManager[3] can restrict the access to certain sensitive APIs, JavaScript does not provide any kind of privilege separation between code loaded from different packages. That is, any third-party package has the full privileges of the entire application. This situation is compounded by the fact that many npm packages run outside of a browser, in particular on the Node.js platform, which does not provide any kind of sandbox.

3 https://docs.oracle.com/javase/6/docs/api/java/lang/SecurityManager.html

Instead, any third-party package can access, e.g., the file system and the network.

NO SYSTEMATIC VETTING The process of discovering vulnerabilities in npm packages is still in its infancy. There currently is no systematic vetting process for code published on npm. Instead, known vulnerabilities are mostly reported by individuals, who find them through manual analysis or in recent research work, e.g., injection vulnerabilities [SPL18], regular expression denial of service [Dav+18; SP18], path traversals [Gon18], binding layer bugs [Bro+17].

PUBLISHING MODEL In order to publish a package, a developer needs to first create an account on the npm website. Once this prerequisite is met, adding a new package to the repository is as simple as running the "npm publish" command in a folder containing a package.json file. The user who first published the package is automatically added to the maintainers set and hence she can release future versions of that package. She can also decide to add additional npm users as maintainers. What is interesting to notice about this model is that it does not require a link to a public version control system, e.g., GitHub, hosting the code of the package. Nor does it require that persons who develop the code on such external repositories also have publishing rights on npm. This disconnect between the two platforms has led to confusion[4] in the past and to stealthy attacks that target npm accounts without changes to the versioning system.

2.2.2 *Threat Models*

The idiosyncratic security properties of npm, as described above, enable several scenarios for attacking users of npm packages. The following discusses threat models that either correspond to attacks that have already occurred or that we consider to be possible in the future.

MALICIOUS PACKAGES (TM-MAL) Adversaries may publish packages containing malicious code on npm and hence trick other users into installing or depending on such packages. In 2018, the eslint-scope incident mentioned earlier has been an example of this threat. The package deployed its payload at installation time through an automatically executed

4 http://www.cs.tufts.edu/comp/116/archive/spring2018/etolhurst.pdf

post-installation script. Other, perhaps more stealthy methods for hiding the malicious behavior could be envisioned, such as downloading and executing payloads only at runtime under certain conditions. Strongly related to malicious packages are packages that violate the user's privacy by sending usage data to third parties, e.g., insight[5] or analytics-node[6]. While these libraries are legitimate under specific conditions, some users may not want to be tracked in this way. Even though the creators of these packages clearly document the tracking functionality, transitive dependents may not be aware that one of their dependencies deploys tracking code.

EXPLOITING UNMAINTAINED LEGACY CODE (TM-LEG) As with any larger code base, npm contains vulnerable code, some of which is documented in public vulnerability databases such as npm security advisories[7] or Snyk vulnerability DB[8]. As long as a vulnerable package remains unfixed, an attacker can exploit it in applications that transitively depend on the vulnerable code. Because packages may become abandoned due to developers inactivity [CM17] and because npm does not offer a forking mechanism, some packages may never be fixed. Even worse, the common practice of locking dependencies may prevent applications from using fixed versions even when they are available.

PACKAGE TAKEOVER (TM-PKG) An adversary may convince the current maintainers of a package to add her as a maintainer. For example, in the recent *event-stream* incident[9], the attacker employed social engineering to obtain publishing rights on the target package. The attacker then removed the original maintainer and hence became the sole owner of the package. A variant of this attack is when an attacker injects code into the source base of the target package. For example, such code injection may happen through a pull request, via compromised development tools, or even due to the fact that the attacker has commit rights on the repository of the package, but not npm publishing rights. Once vulnerable or malicious code is injected, the legitimate maintainer would publish the package on npm, unaware of its security problems. Another takeover-like attack is typosquatting, where an adversary publishes malicious code under a package name similar to the name of a legitimate, popular package. Whenever a user accidentally

5 https://www.npmjs.com/package/insight
6 https://www.npmjs.com/package/analytics-node
7 https://www.npmjs.com/advisories
8 https://snyk.io/vuln/?type=npm
9 https://github.com/dominictarr/event-stream/issues/116

mistypes a package name during installation, or a developer mistypes the name of a package to depend on, the malicious code will be installed. Previous work shows that typosquatting attacks are easy to deploy and effective in practice [Tsc16].

ACCOUNT TAKEOVER (TM-ACC) The security of a package depends on the security of its maintainer accounts. An attacker may compromise the credentials of a maintainer to deploy insecure code under the maintainer's name. At least one recent incident (eslint-scope) is based on account takeover. While we are not aware of how the account was hijacked in this case, there are various paths toward account takeover, e.g., weak passwords, social engineering, reuse of compromised passwords, and data breaches on npm.

COLLUSION ATTACK (TM-COLL) The above scenarios all assume a single point of failure. In addition, the npm ecosystem may get attacked via multiple instances of the above threats. Such a collusion attack may happen when multiple maintainers decide to conspire and to cause intentional harm, or when multiple packages or maintainers are taken over by an attacker.

2.3 METHODOLOGY

To analyze how realistic the above threats are, we systematically study package dependencies, maintainers, and known security vulnerabilities in npm. The following explains the data and metrics we use for this study.

2.3.1 *Data Used for the Study*

PACKAGES AND THEIR DEPENDENCIES To understand the impact of security problems across the ecosystem, we analyze the dependencies between packages and their evolution.

Definition 2.3.1 *Let t be a specific point in time, P_t be a set of npm package names, and $E_t = \{(p_i, p_j) | p_i \neq p_j \in P_t\}$ a set of directed edges between packages, where p_i has a regular dependency on p_j. We call $G_t = (P_t, E_t)$ the* ***npm dependency graph*** *at a given time t.*

We denote the universe of all packages ever published on npm with $\mathcal{P}$. By aggregating the meta information about packages, we can easily con-

struct the dependency graph without the need to download or install every package. Npm offers an API endpoint for downloading this metadata for all the releases of all packages ever published. In total we consider 676,539 nodes and 4,543,473 edges.

To analyze the evolution of packages we gather data about all their releases. As a convention, for any time interval t, such as years or months, we denote with t the snapshot at the beginning of that time interval. For example, G_{2015} refers to the dependency graph at the beginning of the year 2015. In total we analyze 5,386,239 releases, therefore an average of almost eight versions per package. Our observation period ends in April 2018.

MAINTAINERS Every package has one or more developers responsible for publishing updates to the package.

Definition 2.3.2 *For every $p \in P_t$, the set of **maintainers** $M(p)$ contains all users that have publishing rights for p.*

Note that a specific user may appear as the maintainer of multiple packages and that the union of all maintainers in the ecosystem is denoted with $\mathcal{M}$.

VULNERABILITIES The npm community issues advisories or public reports about vulnerabilities in specific npm packages. These advisories specify if there is a patch available and which releases of the package are affected by the vulnerability.

Definition 2.3.3 *We say that a given package $p \in \mathcal{P}$ is **vulnerable at a moment** t if there exists a public advisory for that package and if no patch was released for the described vulnerability at an earlier moment $t' < t$.*

We denote the set of vulnerable packages with $\mathcal{V} \subset \mathcal{P}$. In total, we consider 609 advisories affecting 600 packages. We extract the data from the publicly available npm advisories[10].

2.3.2 *Metrics*

We introduce a set of metrics for studying the risk of attacks on the npm ecosystem.

10 https://www.npmjs.com/advisories

PACKAGES AND THEIR DEPENDENCIES The following measures the influence of a given package on other packages in the ecosystem.

Definition 2.3.4 *For every $p \in P_t$, the **package reach** $\mathrm{PR}(p)$ represents the set of all the packages that have a transitive dependency on p in G_t.*

Note that the package itself is not included in this set. The reach $\mathrm{PR}(p)$ contains names of packages in the ecosystem. Therefore, the size of the set is bounded by the following values $0 \leq |\mathrm{PR}(p)| < |P_t|$.

Since $|\mathrm{PR}(p)|$ does not account for the ecosystem changes, the metric may grow simply because the ecosystem grows. To address this, we also consider the average package reach:

$$\overline{\mathrm{PR}_t} = \frac{\sum_{\forall p \in P_t} |\mathrm{PR}(p)|}{|P_t|} \tag{2.1}$$

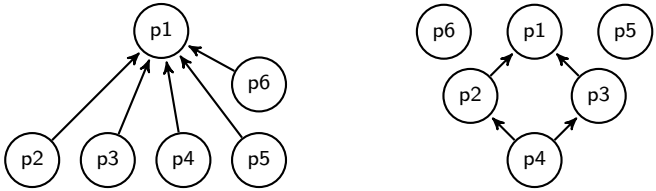
Using the bounds discussed before for $\mathrm{PR}(p)$, we can calculate the ones for its average $0 \leq \overline{\mathrm{PR}_t} < |P_t|$. The upper limit is obtained for a fully connected graph in which all packages can reach all the other packages and hence $|PR(p)| = |P_t| - 1, \forall p$. If $\overline{\mathrm{PR}_t}$ grows monotonously, we say that the ecosystem is getting more dense, and hence the average package influences an increasingly large number of packages.

The inverse of package reach is a metric to quantify how many packages are implicitly trusted when installing a particular package.

Definition 2.3.5 *For every $p \in P_t$, the set of **implicitly trusted packages** $\mathrm{ITP}(p)$ contains all the packages p_i for which $p \in \mathrm{PR}(p_i)$.*

Similarly to the previous case, we also consider the size of the set $|\mathrm{ITP}(p)|$ and the average number of implicitly trusted package $\overline{\mathrm{ITP}_t}$, having the same bounds as their package reach counterpart.

Even though the average metrics $\overline{\mathrm{ITP}_t}$ and $\overline{\mathrm{PR}_t}$ are equivalent for a given graph, the distinction between their non-averaged counterparts is very important from a security point of view. To see why, consider the example in Figure 2.1. The average $\overline{PR} = \overline{ITP}$ is 5/6 = 0.83 both on the right and on the left. However, on the left, a popular package $p1$ is dependent upon by many others. Hence, the package reach of $p1$ is five, and the number of implicitly trusted packages is one for each of the other packages. On the right, though, the number of implicitly trusted packages for $p4$ is three, as users of $p4$ implicitly trust packages $p1$, $p2$, and $p3$.



(a) Wide distribution of trust: $max(\text{PR}) = 5, max(\text{ITP}) = 1$

(b) Narrow distribution of trust: $max(\text{PR}) = 3, max(\text{ITP}) = 3$

FIGURE 2.1: Dependency graphs with different maximum package reaches (PR) and different maximum numbers of trusted packages (ITP).

MAINTAINERS The number of implicitly trusted packages or the package reach are important metrics for reasoning about TM-pkg, but not about TM-acc. That is because users may decide to split their functionality across multiple micropackages for which they are the sole maintainers. To put it differently, a large attack surface for TM-pkg does not imply one for TM-acc.

Therefore, we define maintainer reach $\text{MR}_t(m)$ and implicitly trusted maintainers $\text{ITM}_t(p)$ for showing the influence of maintainers.

Definition 2.3.6 *Let m be an npm maintainer. The* ***maintainer reach*** $\text{MR}(m)$ *is the combined reach of all the maintainer's packages,* $\text{MR}(m) = \cup_{m \in M(p)} \text{PR}(p)$

Definition 2.3.7 *For every $p \in P_t$, the set of* ***implicitly trusted maintainers*** $\text{ITM}(p)$ *contains all the maintainers that have publishing rights on at least one implicitly trusted package,* $\text{ITM}(p) = \cup_{p_i \in \text{ITP}(p)} M(p_i)$.

The above metrics have the same bounds as their packages counterparts. Once again, the distinction between the package and the maintainer-level metrics is for shedding light on the security relevance of human actors in the ecosystem.

Furthermore, to approximate the maximum damage that colluding maintainers can incur on the ecosystem (TM-coll), we define an order in which the colluding maintainers are selected:

Definition 2.3.8 *We call an ordered set of maintainers $L \subset \mathcal{M}$ a* ***desirable collusion strategy*** *iff $\forall m_i \in L$ there is no $m_k \neq m_i$ for which $\cup_{j<i} \text{MR}(m_j) \cup \text{MR}(m_i) < \cup_{j<i} \text{MR}(m_j) \cup \text{MR}(m_k)$.*

Therefore, the desirable collusion strategy is a hill climbing algorithm in which at each step we choose the maintainer that provides the highest

local increase in package reach at that point. We note that the problem of finding the set of n maintainers that cover the most packages is an NP-hard problem called *maximum coverage problem*. Hence, we believe that the proposed solution is a good enough approximation that shows how vulnerable the ecosystem is to a collusion attack, but that does not necessary yield the optimal solution.

VULNERABILITIES For reasoning about TM-leg, we need to estimate how much of the ecosystem depends on vulnerable code:

Definition 2.3.9 *Given all vulnerable packages $p_i \in \mathcal{V}_t$ at time t, we define the **reach of vulnerable code** at time t as* $\mathrm{VR}_t = \cup_{p_i \in \mathcal{V}_t} \mathrm{PR}(p_i)$.

Of course the actual reach of vulnerable code can not be fully calculated since it would rely on *all* vulnerabilities present in npm modules, not only on the published ones. However, since in TM-leg we are interested in publicly known vulnerabilities, we define our metric according to this scenario. In these conditions, the speed at which vulnerabilities are reported is an important factor to consider:

Definition 2.3.10 *Given all vulnerable packages $p_i \in \mathcal{V}_t$ at time t, we define the **vulnerability reporting rate** VRR_t at time t as* $\mathrm{VRR}_t = \frac{|\mathcal{V}_t|}{|P_t|}$.

2.4 RESULTS

We start by reporting the results on the nature of package level dependencies and their evolution over time (corresponding to TM-mal and TM-pkg). We then discuss the influence that maintainers have in the ecosystem (related to TM-acc and TM-coll). Finally, we explore the dangers of depending on unpatched security vulnerabilities (addressing TM-leg).

2.4.1 *Dependencies in the Ecosystem*

To set the stage for a thorough analysis of security risks entailed by the structure of the npm ecosystem, we start with a general analysis of npm and its evolution. Since its inception in 2010, the npm ecosystem has grown from a small collection of packages maintained by a few people to the world's largest software ecosystem. Figure 2.2 shows the evolution of the number of packages available on npm and the number of maintainers responsible for these packages. Both numbers have been increasing super-

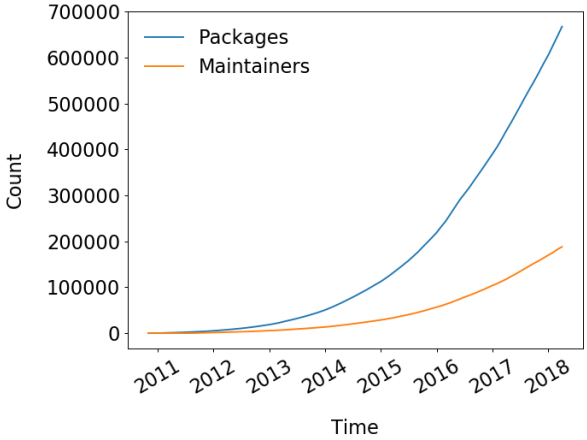


FIGURE 2.2: Evolution of number of packages and maintainers.

linearly over the past eight years. At the end of our measurement range, there is a total of 676,539 packages, a number likely to exceed one million in the near future. These packages are taken care of by a total of 199,327 maintainers. The ratio of packages to maintainers is stable across our observation period (ranging between 2.81 and 3.51).

In many ways, this growth is good news for the JavaScript community, as it increases the code available for reuse. However, the availability of many packages may also cause developers to depend on more and more third-party code, which increases the attack surface for TM-pkg by giving individual packages the ability to impact the security of many other packages. The following analyzes how the direct and transitive dependencies of packages are evolving over time (Section 2.4.1.1) and how many other packages individual packages reach via dependencies (Section 2.4.1.2).

2.4.1.1 *Direct and Transitive Dependencies*

Figure 2.3 shows how many other packages an average npm package depends on directly and transitively. The number of direct dependencies has been increasing slightly from 1.3 in 2011 to 2.8 in 2018, which is perhaps unsurprising given the availability of an increasing code base to reuse. The less obvious observation is that a small, linear increase in direct dependencies leads to a significant, super-linear increase in transitive dependencies. As shown by the upper line in Figure 2.3, the number of transitive depen-

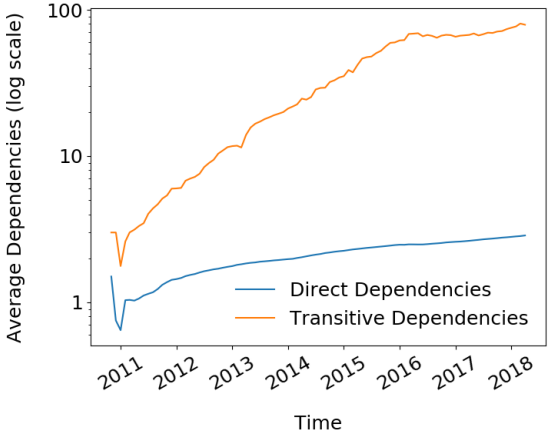


FIGURE 2.3: Evolution of direct package dependencies and its impact on transitive dependencies. Note the logarithmic scale on the y-axis.

dencies of an average package has increased to a staggering 80 in 2018 (note the logarithmic scale).

From a security perspective, it is important to note that each directly or transitively depended on package becomes part of the implicitly trusted code base. When installing a package, each depended upon package runs its post-installation scripts on the user's machine – code executed with the user's operating system-level permissions. When using the package, calls into third-party modules may execute any of the code shipped with the depended upon packages.

One can observe in Figure 2.3 a chilling effect on the number of dependencies around the year 2016 which will become more apparent in the following graphs. Decan et al. [DMG19] hypothesize that this effect is due to the left-pad incident. In order to confirm that this is not simply due to removal of more than a hundred packages belonging to the left-pad's owner, we remove all the packages owned by this maintainer. We see no significant difference for the trend in Figure 2.3 when removing these packages, hence we conclude that indeed there is a significant change in the structure of transitive dependencies in the ecosystem around 2016.

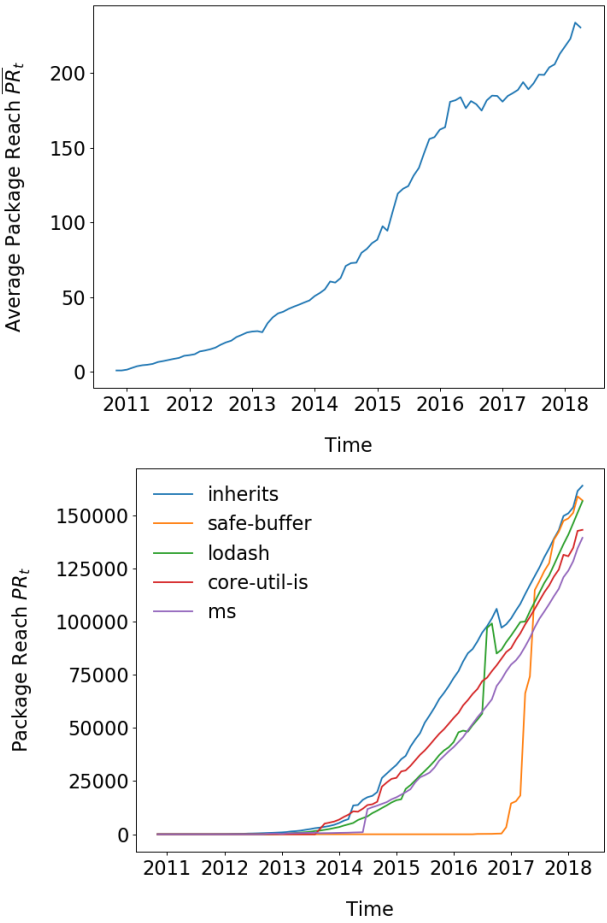


FIGURE 2.4: Evolution of package reach for an average package (top) and the top-5 packages (bottom).

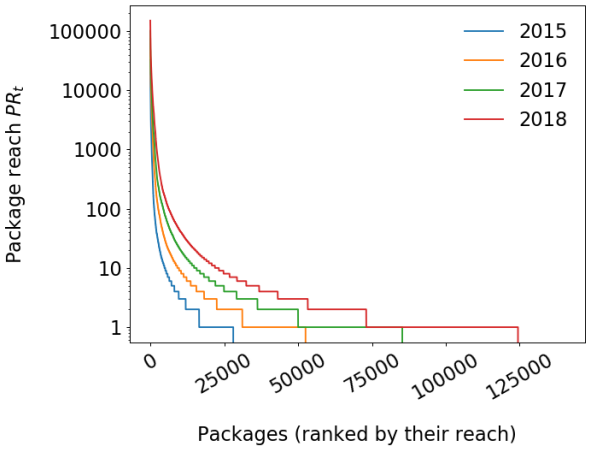


FIGURE 2.5: Distribution of package reach by individual packages, and how it changes over time. Note the log scale on the vertical axis.

2.4.1.2 *Package Reach*

The above analysis focuses on depended upon packages. We now study the inverse phenomenon: packages impacted by individual packages, i.e., package reach as defined in Section 2.3. Figure 2.4 shows how many other packages a single package reaches via direct or indirect dependencies. The graph at the top is for an average package, showing that it impacts about 230 other packages in 2018, a number that has been growing since the creation of npm. The graph at the bottom shows the package reach of the top-5 packages (top in terms of their package reach, as of 2018). In 2018, these packages each reach between 134,774 and 166,086 other packages, making them an extremely attractive target for attackers.

To better understand how the reach of packages evolves over time, Figure 2.5 shows the distribution of reached packages for multiple years. For example, the red line shows that in 2018, about 24,500 packages have reached at least 10 other packages, whereas only about 9,500 packages were so influential in 2015. Overall, the figure shows that more and more packages are reaching a significant number of other packages, increasing the attractiveness of attacks that rely on dependencies.

The high reach of a package amplifies the effect of both vulnerabilities (TM-leg) and of malicious code (TM-mal). As an example for the latter, consider the event-stream incident discussed when introducing TM-acc in

Section 2.2.2. By computing event-stream's reach and comparing it with other packages, we see that this package is just one of many possible targets. As of April 1, 2018 (the end of our measurement period), event-stream has a reach of 5,466. That is, the targeted package is relatively popular, but still far from being the top-most attractive package to compromise. In fact, 1,165 other packages have a greater or equal reach than event-stream.

In order to perform a similar analysis for the eslint-scope security incident, we need to use a slightly modified version of package reach. This attack targeted a development tool, namely eslint, hence, to fully estimate the attack surface we need to consider dev dependencies in our definition of reach. We do not normally consider this type of dependencies in our measurements because they are not automatically installed with a package, unlike regular dependencies. They are instead used only by the developers of the packages. Therefore the modified version of package reach considers both transitive regular dependencies and direct dev dependencies.

We observe that eslint-scope has a modified reach of more than 100,000 packages at the last observation point in the data set. However, there are 347 other packages that have a higher reach, showing that even more serious attacks may occur in the future.

2.4.2 *Analysis of Maintainers*

We remind the reader that there is a significant difference between npm maintainers and repository contributors, as discussed in Section 2.2.1. Even though contributors also have a lot of control over the code that will eventually end up in an npm package, they can not release a new version on npm, only the maintainers have this capability. Hence, the discussion that follows, about the security risks associated with maintainers, should be considered a lower bound for the overall attack surface.

Attacks corresponding to TM-acc in which maintainers are targeted are not purely hypothetical as the infamous eslint-scope incident discussed earlier shows. In this attack, a malicious actor hijacked the account of an influential maintainer and then published a version of eslint-scope containing malicious code. This incident is a warning for how vulnerable the ecosystem is to targeted attacks and how maintainers influence can be used to deploy malware at scale. We further discuss the relation between packages and maintainers.

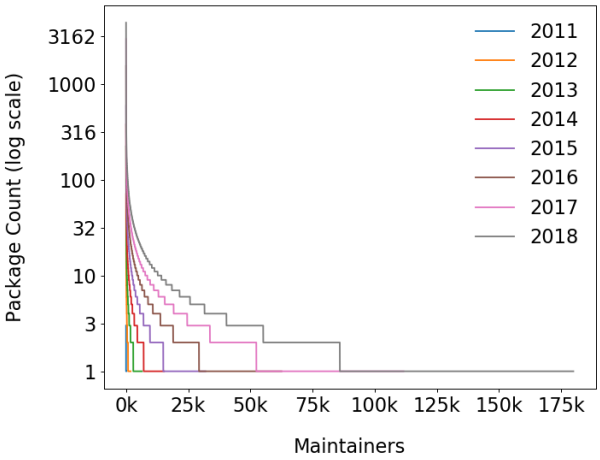


FIGURE 2.6: Evolution of maintainers sorted by package count per year.

2.4.2.1 *Packages per Maintainer*

Even though the size of the ecosystem grows super-linearly as discussed in Section 2.4.1, one would expect that this is caused mainly by new developers joining the ecosystem. However, we observe that the number of packages per maintainer also grows suggesting that the current members of the platform are actively publishing new packages. The average number of packages controlled by a maintainer raises from 2.5 in 2012 to 3.5 in 2013 and almost 4.5 in 2018. Conversely, there are on average 1.35 maintainers in the lifetime of a package. The top 5,000 most popular packages have an average number of 2.83 maintainers. This is not unexpected, since multiple people are involved in developing the most popular packages, while for the majority of new packages there is only one developer.

Next, we study in more detail the evolution of the number of packages a maintainer controls. Figure 2.6 shows the maintainer package count plotted versus the number of maintainers having such a package count. Every line represents a year. The scale is logarithmic to base 10. It shows that the majority of maintainers maintain few packages, yet some maintainers maintain over 100 packages. Over the years, the package count for the maintainers increased consistently. In 2015, only slightly more than 25,000 maintainers maintained more than one package, whereas this number has more than tripled by 2018.

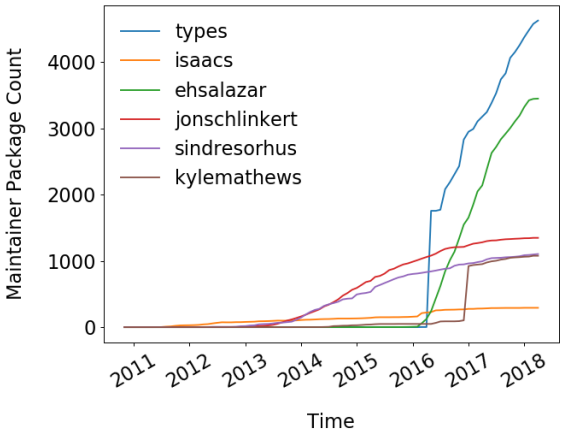


FIGURE 2.7: Evolution of package count for six popular maintainers.

We further analyze five different maintainers in top 20 according to number of packages and plot the evolution of their package count over the years in Figure 2.7. *types* is the largest maintainer of type definitions for TypeScript, most likely a username shared by multiple developers at Microsoft, *ehsalazar* maintains many security placeholder packages, *jonschlinkert* and *sindresorhus* are maintaining many micropackages and *isaacs* is the npm founder. From Figure 2.7 we can see that for two of these maintainers the increase is superlinear or even near exponential: *types* and *kylemathews* have sudden spikes where they added many packages in a short time. We explain this by the tremendous increase in popularity for TypeScript in the recent years and by the community effort to prevent typosquatting attacks by reserving multiple placeholder. The graph of the other maintainers is more linear, but surprisingly it shows a continuous growth for all the six maintainers.

2.4.2.2 *Implicitly Trusted Maintainers*

One may argue that the fact that maintainers publish new packages is a sign of a healthy ecosystem and that it only mimics its overall growth. However, we show that while that may be true, we also see an increase in the general influence of maintainers. That is, on average every package tends to transitively rely on more and more maintainers over time.

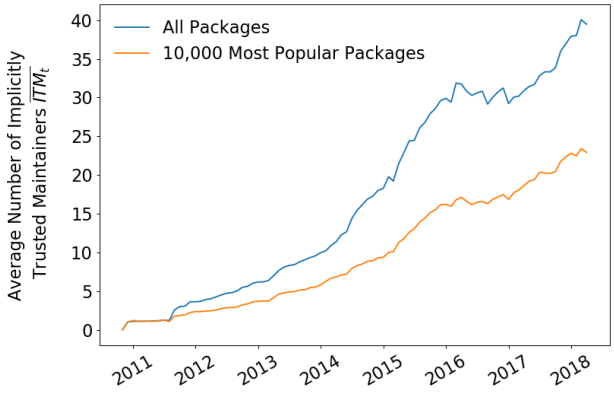


FIGURE 2.8: Evolution of average number of implicitly trusted maintainers over years in all packages and in the most popular ones.

In Figure 2.8 we show the evolution of $\overline{ITM_t}$, the average number of implicitly trusted maintainers. As can be seen, $\overline{ITM_t}$ almost doubled in the last three years for the average npm package, despite the plateau of the curve reached in 2016 which we again speculate it is caused by the left-pad incident. This is a worrisome development since compromising any of the maintainer accounts a package trusts may seriously impact the security of that package, as discussed in TM-acc. The positive aspect of the data in Figure 2.8 is that the growth in the number of implicitly trusted maintainers seems to be less steep for the top 10,000 packages compared to the whole ecosystem. We hypothesize that the developers of popular packages are aware of this problem and actively try to limit the $\overline{ITM_t}$. However, a value over 20 for the average popular package is still high enough to be problematic.

When breaking the average $\overline{ITM_t}$ discussed earlier into individual points in Figure 2.9, one can observe that the majority of these packages can be influenced by more than one maintainer. This is surprising since most of the popular packages are micropackages such as "inherits" or "left-pad" or libraries with no dependencies like "moment" or "lodash". However, only around 30% of these top packages have a maintainer cost higher than 10. Out of these, though, there are 643 packages influenced by more than a hundred maintainers.

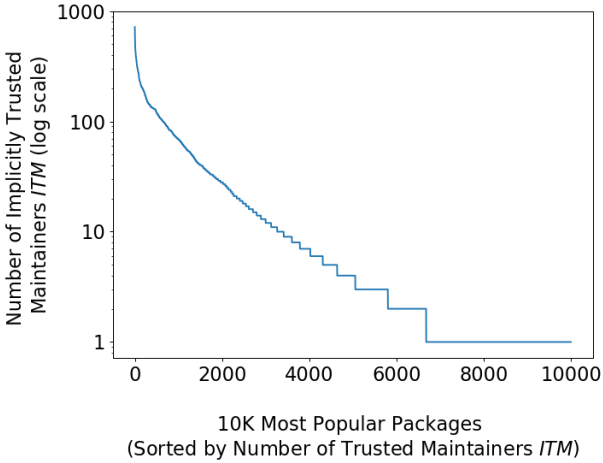


FIGURE 2.9: Number of implicitly trusted maintainers for top 10,000 most popular packages.

2.4.2.3 *Maintainers Reach*

In Figure 2.10, we plot the reach MR_t of the maintainers in the npm ecosystem. The reach has increased over the years at all levels. For example, in 2015 there were 2,152 maintainers that could affect more than 10 packages, and this number increased to 4,041 in 2016, 6,680 in 2017 and finally reaching an astonishingly high 10,534 in 2018. At the other end of the distribution, there were 59 maintainers that could affect more than 10,000 packages in 2015, 163 in 2016, 249 in 2017 and finally 391 in 2018. The speed of growth for MR_t is worrisome, showing that more and more developers have control over thousands of packages. If an attacker manages to compromise the account of any of the 391 most influential maintainers, the community will experience a serious security incident, reaching twice as many packages as in the event-stream attack.

Finally, we look at the scenario in which multiple popular maintainers collude, according to the desirable collusion strategy introduced in Section 2.3.2, to perform a large-scale attack on the ecosystem, i.e., TM-col. In Figure 2.11 we show that 20 maintainers can reach more than half of the ecosystem. Past that point every new maintainer joining does not increase significantly the attack's performance.

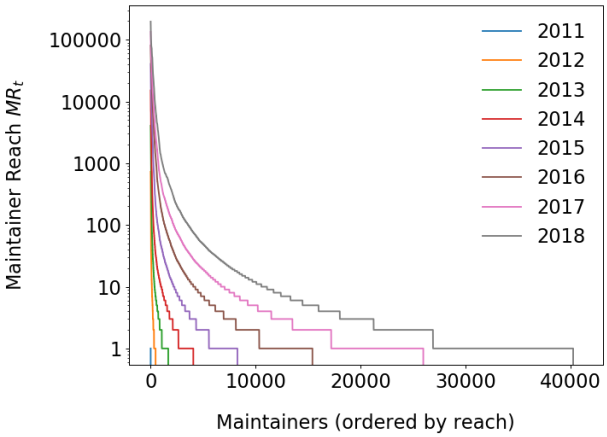


FIGURE 2.10: Distribution of maintainers reach in different years.

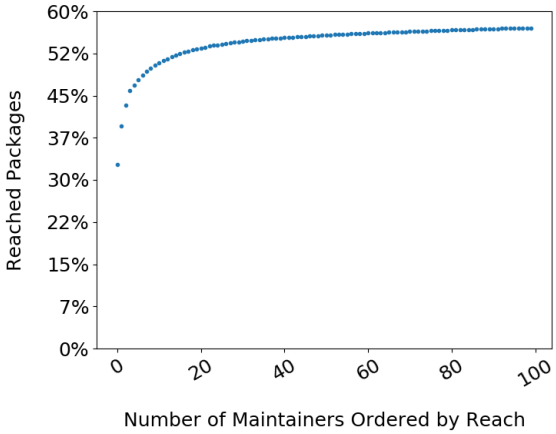


FIGURE 2.11: Combined reach of 100 influential maintainers.

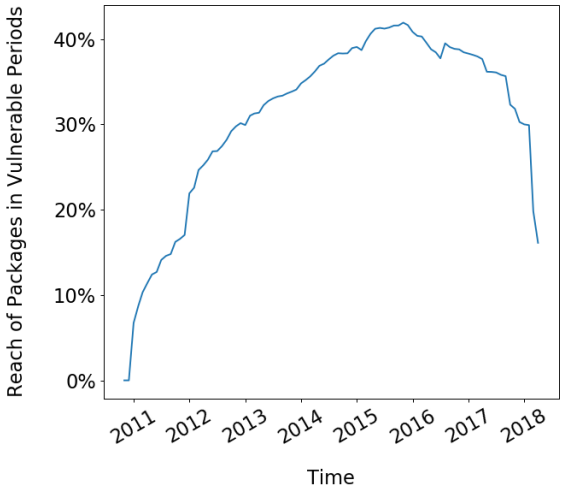


FIGURE 2.12: Total reach of packages for which there is at least one unpatched advisory (vulnerability reach VR_t).

2.4.3 *Security Advisories Evolution*

Next, we study how often vulnerabilities are reported and fixed in the npm ecosystem (TM-leg). Figure 2.13 shows the number of reported vulnerabilities in the lifetime of the ecosystem. The curve seems to resemble the evolution of number of packages presented in Figure 2.2, with a steep increase in the last two years. To explore this relation further we plot in Figure 2.14 the evolution of the number of advisories reported per 10,000 packages and we observe that it grows from two in 2013 to almost eight in 2018. This is a sign of a healthy security community that reports vulnerabilities at a very good pace, keeping up with the growth of the ecosystem.

When analyzing the type of reported vulnerabilities in details, we observe that almost half of the advisories come from two large-scale campaigns and not a broader community effort: First, there are 141 advisories published in January 2017 that involve npm packages that download resources over HTTP, instead of HTTPs. Second, there are 120 directory traversal vulnerabilities reported as part of the research efforts of Liang Gong [Gon18]. Nevertheless, this shows the feasibility of large-scale vulnerability detection and reporting on npm.

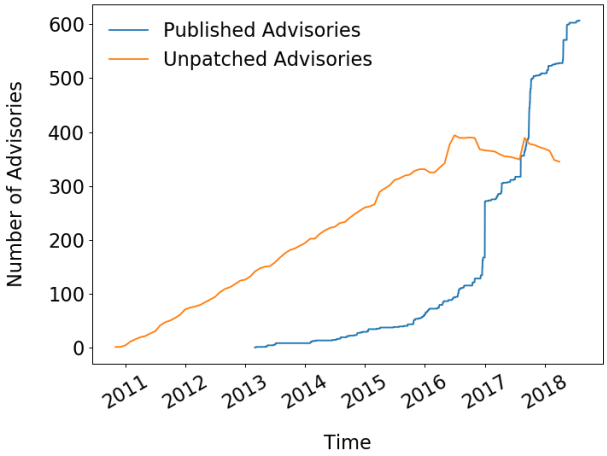


FIGURE 2.13: Evolution of the total and unpatched number of advisories.

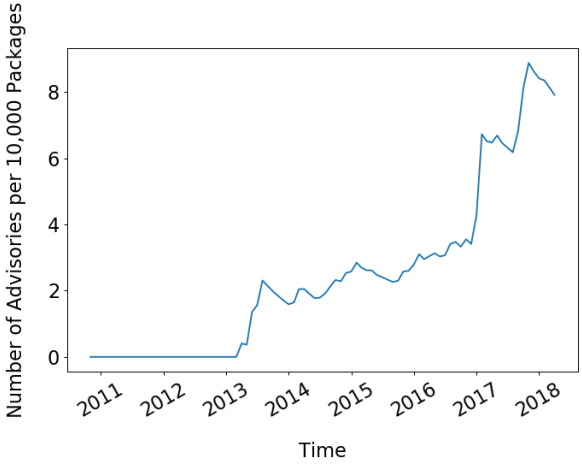


FIGURE 2.14: Evolution of VRR_t, the rate of published vulnerabilities per 10,000 packages.

Publishing an advisory helps raise awareness of a security problem in an npm package, but in order to keep the users secure, there needs to be a patch available for a given advisory. In Figure 2.13 we show the evolution of the number of unpatched security vulnerabilities in npm, as defined in Section 2.3. This trend is alarming, suggesting that two out of three advisories are still unpatched, leaving the users at risk. When manually inspecting some of the unpatched advisories we notice that a large percentage of unpatched vulnerabilities are actually advisories against malicious typosquatting packages for which no fix can be available.

To better understand the real impact of the unpatched vulnerabilities we analyze how much of the ecosystem they impact, i.e., vulnerability reach as introduced in Section 2.3.2. To that end, we compute the reach of unpatched packages at every point in time in Figure 2.12. At a first sight, this data shows a much less grim picture than expected, suggesting that the reach of vulnerable packages is dropping over time. However, we notice that the effect of vulnerabilities tends to be retroactive. That is, a vulnerability published in 2015 affects multiple versions of a package released prior to that date, hence influencing the data points corresponding to the years 2011-2014 in Figure 2.12. Therefore, the vulnerabilities that will be reported in the next couple of years may correct for the downwards trend we see on the graph. Independent of the downwards trend, the fact that for the majority of the time the reach of vulnerable unpatched code is between 30% and 40% is alarming.

2.5 POTENTIAL MITIGATIONS

The following section discusses ideas for mitigating some of the security threats in the npm ecosystem. We do not provide here fully developed solutions, but instead outline ideas for future research, along with an initial assessment of their potential and challenges involved in implementing them.

2.5.1 *Raising Developer Awareness*

One line of defense against the attacks described in this chapter is to make developers who use third-party packages more aware of the risks entailed by depending on a particular package. Currently, npm shows for each package the number of downloads, dependencies, dependents, and open issues in the associated repository. However, the site does not show

FIGURE 2.15: Correlation between number of vulnerabilities and number of dependencies.

any information about the transitive dependencies or about the number of maintainers that may influence a package, i.e., our ITP and ITM metrics. As initial evidence that including such metrics indeed predicts the risk of security issues, Figure 2.15 shows the number of implicitly trusted packages versus the number of vulnerabilities a package is affected by. We find that the two values are correlated (Pearson correlation coefficient of 0.495), which is not totally unexpected since adding more dependencies increases the chance of depending on vulnerable code. Showing such information, e.g., the ITP metric, could help developers make more informed decisions about which third-party packages to rely on.

2.5.2 *Warning about Vulnerable Packages*

To warn developers about unpatched vulnerabilities in their dependencies, the `npm audit` tool has been introduced. It compares all directly depended upon packages against a database of known vulnerabilities, and warns a developer when depending upon a vulnerable version of a package. While being a valuable step forward, the tool currently suffers from at least three limitations. First, it only considers direct dependencies but ignores any vulnerabilities in transitive dependencies. Second, the tool is limited to known vulnerabilities, and hence its effectiveness depends on how fast advisories are published. Finally, this defense is insufficient against malware attacks.

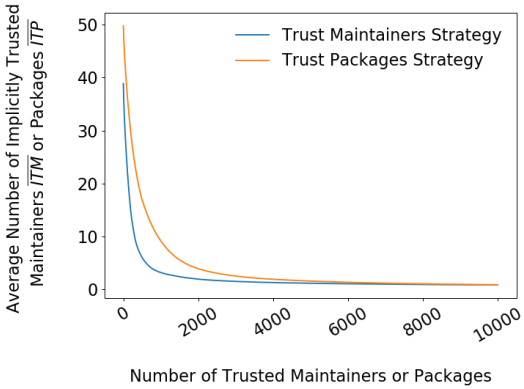


FIGURE 2.16: Decrease in average number of implicitly trusted maintainers and packages as the set of trusted maintainers or packages increases.

2.5.3 *Code Vetting*

A proactive way of defending against both vulnerable and malicious code is code vetting. Similar to other ecosystems, such as mobile app stores, whenever a new release of a vetted package is published, npm could analyze its code. If and only if the analysis validates the new release, it is made available to users. Since the vetting process may involve semi-automatic or even manual steps, we believe that it is realistic to assume that it will be deployed step by step in the ecosystem, starting with the most popular packages. Figure 2.16 (orange curve) illustrates the effect that such code vetting could have on the ecosystem. The figure shows how the average number of implicitly trusted packages, $\overline{\text{ITP}}$, reduces with an increase in number of vetted and therefore trusted packages. For example, vetting the most dependent upon 1,500 packages would reduce the $\overline{\text{ITP}}$ ten fold, and vetting 4,000 packages would reduce it by a factor of 25.

An obvious question is how to implement such large-scale code vetting, in particular, given that new versions of packages are released regularly. To estimate the cost of vetting new releases, Figure 2.17 shows the average number of lines of code that are changed per release of a package, and would need to be vetted to maintain a specific number of trusted packages. For example, vetting the changes made in a single new release of the top 400 most popular packages requires to analyze over 100,000 changed lines of code. One way to scale code vetting to this amount of code could be automated code analysis tools. There are several recent efforts for im-

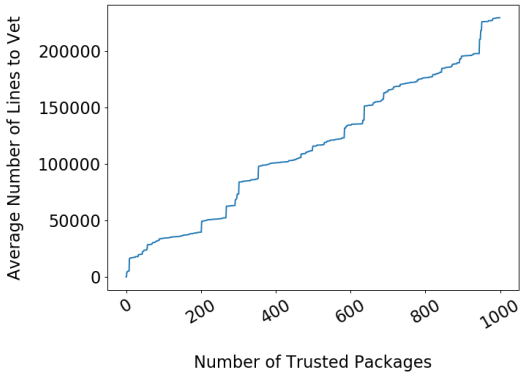


FIGURE 2.17: Number of lines of code that need to be vetted for achieving a certain number of trusted packages.

proving the state of the art npm security auditing, both from academia, e.g., SYNODE in Chapter 7, BreakApp [Vas+18], NodeSec [Gon18], NoRegrets [MMT18], Node.cure [DWL18], and from industry practitioners, e.g., Semmle[11], r2c[12], and DeepScan[13]. Orthogonal to automated code analysis tools, the community could establish crowd-sourced package vetting, e.g., in a hierarchically organized code distribution model similar to the Debian ecosystem.

Another challenge for code vetting is that npm packages, in contrast to apps in mobile app stores, are used across different platforms with different security models. For example, XSS vulnerabilities are relevant only when a package is used on the client-side, whereas command injection via the `exec` API, as discussed in Chapter 4, is a concern only on the server-side. A code vetting process could address this challenge by assigning platform-specific labels, e.g., "vetted for client-side" or "vetted for server-side", depending on which potential problems the vetting reveals.

2.5.4 *Training and Vetting Maintainers*

Another line of proactive defense could be to systematically train and vet highly influential maintainers. For example, this process could validate

11 https://semmle.com/
12 https://r2c.dev/
13 https://deepscan.io/

the identity of maintainers, support maintainers in understanding basic security principles, and ensure that their accounts are protected by state-of-the-art techniques, such as two-factor authentication. To assess the effect that such a process would have, we simulate how training and vetting a particular number of *trusted maintainers* influences the average number of implicitly trusted maintainers, $\overline{\text{ITM}}$. The simulation assumes that the most influential maintainers are vetted first, and that once a maintainer is vetted she is ignored in the computation of the $\overline{\text{ITM}}$. The results of this simulation (Figure 2.16) show a similar effect as for vetting packages: Because some maintainers are highly influential, vetting a relatively small number of maintainers can significantly reduce security risks. For example, vetting around 140 maintainers cuts down the $\overline{\text{ITM}}$ in half, and vetting around 600 could even reduce $\overline{\text{ITM}}$ to less than five. These results show that this mechanism scales reasonably well, but that hundreds of maintainers need to be vetted to bring the average number of implicitly trusted maintainers to a reasonable level. Moreover, two-factor authentication has its own risks, e.g., when developers handle authentication tokens in an insecure way[14] or when attackers attempt to steal such tokens, as in the eslint-scope incident.

2.6 CONCLUSIONS

In this chapter, we present a large-scale study of security threats resulting from the densely connected structure of npm packages and maintainers. The overall conclusion is that npm is a small world with high risks. It is "small" in the sense that packages are densely connected via dependencies. The security risk are "high" in the sense that vulnerable or malicious code in a single package may affect thousands of others, and that a single misbehaving maintainer, e.g., due to a compromised account, may have a huge negative impact. These findings show that recent security incidents in the npm ecosystem are likely to be the first signs of a larger problem, and not only unfortunate individual cases. To mitigate the risks imposed by the current situation, we analyze the potential effectiveness of several mitigation strategies. We find that trusted maintainers and a code vetting process for selected packages could significantly reduce current risks.

14 https://blog.npmjs.org/post/182015409750/automated-token-revocation-for-when-you

3

MINIFIED AND OBFUSCATED CODE ON THE WEB

In this chapter we study how widespread code transformations (see particularity P_3 in the introduction) are in real-world, client-side JavaScript code. To that end, we propose a novel methodology based on unsupervised machine learning models that can distinguish transformed code from non-transformed code. While we limit the study to client-side JavaScript, we believe a similar methodology can be used to analyze other parts of full-stack JavaScript applications, i.e., the standalone server-side code or the third-party code. This chapter shares material with the corresponding publication [SSP19].

3.1 MOTIVATION

An effective way to hide the maliciousness of JavaScript code are code transformations that preserve the overall behavior of a script while making it harder to understand and analyze. Such transformations affect both manual code inspection, e.g., because the code becomes harder to understand, and automated code analysis, e.g., because the malicious behavior is disguised as apparently harmless operations. There exist a variety of code transformations, ranging from renaming of local variables to more complex code changes that affect the control flow and data flow. We refer to transformations aimed at reducing code size, typically by renaming local variables to shorter names, as *minification*. In contrast, we refer to more complex transformations aimed at hindering the understanding and analysis of code as *obfuscation*. It is important to note that minification and obfuscation may be used for legitimate reasons, such as reducing code size or protecting intellectual property. However, independently of what the reason for transforming code is, it affects the ability of human and automated security analysis.

Despite the potential impact that minification and obfuscation may have on security analysis, little is currently known about how real-world websites use such transformations. A better understanding of what kinds of code transformations are applied in the wild could guide future efforts on making the web more secure. In particular, knowing how widespread

minification and obfuscation are helps future analyses to focus on relevant problems. Moreover, understanding what kinds of transformation tools are the most popular enables the development of targeted defense techniques. Unfortunately, to the best of our knowledge, there currently is no comprehensive study of code transformations in the web.

This chapter presents a large-scale empirical study of minification and obfuscation in client-side web code. The study involves 967,149 JavaScript files gathered from the top 100,000 websites. We analyze how many of these scripts are transformed through minification and obfuscation, respectively, and which tools are used for these transformations. Moreover, we study which kinds of scripts are transformed particularly often and inspect the runtime behavior of some obfuscated scripts. Finally, we analyze which costs code transformation may incur by assessing to what extent popular transformation tools influence the performance and correctness of JavaScript code.

Given the large-scale nature of our study, we rely, at least in parts, on automation to answer the above questions. To this end, we present a neural network-based classifier that identifies JavaScript code with particular properties. For example, the classifier can be trained to distinguish transformed from non-transformed code, minified from obfuscated code, and to identify code transformed with particular tools. We show that the classification has very high accuracy for these tasks, providing an effective way to identify particular kinds of scripts across all studied websites.

The study addresses six research questions.

RQ1: How prevalent are minification and obfuscation in client-side JavaScript code? Answering this question is important to determine whether code transformations should be considered by security analyses at all, and what kinds of transformations such analyses should focus on. We find that code transformations are very widespread, affecting 38% of all client-side scripts. The majority of the transformed code has been minified, whereas less than 1% of all code is obfuscated. Even though the percentage of obfuscated code is low, the absolute number of obfuscated scripts (2,842) still motivates work on de-obfuscation, as these scripts arguably are the most interesting for security analysis.

RQ2: Which tools are used to obfuscate code in the web? There is a variety of tools available for obfuscating JavaScript code. Understanding which tools and transformation techniques are used most often in practice helps prioritize efforts toward dealing with transformed code. Our study finds that a single obfuscation tool accounts for most obfuscated scripts in the web:

2,551 obfuscated scripts resemble the output of the *Daft Logic Obfuscator*, an online tool that is available free of charge, motivating future work to consider obfuscation techniques implemented by this tool.

RQ3: Does the prevalence of code transformations differ across different kinds of scripts or websites? Since there are many possible reasons for minifying and obfuscating code, it is interesting to ask whether specific kinds of scripts are transformed more often than others. We find that third-party scripts, i.e., code loaded from another website than the one visited by a user, is about twice as likely to be transformed than scripts loaded from the visited website. Possible reasons for this distribution include that content delivery networks minify libraries to reduce network traffic, and that advertisement and tracking code is transformed to protected intellectual property. Studying the prevalence of obfuscation in different categories of websites shows that some categories, e.g., sites with adult content, contain more obfuscated scripts than an average website.

RQ4: What kind of behavior do developers hide behind obfuscation? To further understand the reasons for obfuscating code, we analyze the execution behavior of obfuscated scripts and manually inspect a subset of them. We find that many of the obfuscated scripts access APIs that are typically used for tracking, fingerprinting, cookie syncing, or cookie theft. We also identify a script with an unusually high number of calls to `performance.now`, which could be because the script is exploiting some timing channel.

The final two questions are about potential costs that applying code transformations may incur.

RQ5: How do code transformations affect the performance of code? We find that most obfuscation tools negatively affect performance, i.e., they slow down the execution of the code. In contrast, most minification tools either have no effect on performance or speed up the execution of the code. These findings show that complex code transformations may come at a non-negligible cost, motivating future work on performance-invariant obfuscation.

RQ6: How do code transformations affect the correctness of code? Developers applying an automated code transformation tool may naively assume that the transformation preserves the overall semantics of the code. However, we find that existing tools for both minification and obfuscation often produce corrupt code that behaves differently from the original code. Only about half of the transformed code completely preserves the original semantics, motivating future work on more reliable transformation tools.

The results in this chapter are relevant for at least four groups of people. First, the study provides insights for developers of security-related program analyses, e.g., static malware checkers and de-obfuscators. In particular, we show that transformed code must be considered by any analysis aimed at real-world, client-side JavaScript code, and we show what kinds of transformations are the most important in practice. Second, the study affects developers of obfuscation and minification tools by highlighting the costs that using state-of-the-art tools imply. Third, the study informs users of transformation tools about the effects that using such tools may have on the performance and correctness of code. Finally, the classifier that we develop to conduct our study enables researchers interested in analyzing real-world JavaScript code to focus on code relevant for their research. For example, the classifier can accurately identify obfuscated code among hundreds of thousands of scripts.

In summary, this chapter contributes the following:

- The first large-scale study of minification and obfuscation in real-world, client-side web application code.
- Insights about how code transformations are used in practice, including evidence that minification is widespread, that more complex obfuscation is rather rare yet non-negligible, and that particular obfuscation techniques clearly dominate.
- An automated classification technique that accurately identifies different kinds of transformed code. The technique is useful to select particular scripts, e.g., those with obfuscated code, for further analysis.
- A benchmark of obfuscated JavaScript files gathered from various popular websites, which we make available for future work on de-obfuscating, analyzing, and understanding obfuscated code.

3.2 CLASSIFICATION OF SCRIPTS

Addressing RQ1, RQ2, RQ3, and RQ4 at the scale of hundreds of thousands of scripts requires an automated technique to determine whether a script has been transformed, and if yes, in what way. While a skilled human could manually label files with high accuracy, that approach does not scale to the amount of JavaScript code considered in our study. One approach to address this challenge would be to define a set of heuristics,

e.g., based on idiosyncrasies of specific transformation tools, and to determine the properties of code based on these heuristics. Unfortunately, the heuristics-based approach relies on human expertise for identifying code properties that are unique to specific transformation techniques, and it cannot be easily adapted to other transformation tools. Instead, we address the challenge of determining properties of code through machine learning. This section presents two machine learning-based approaches for classifying JavaScript code. We use the more effective of the two approaches as the basis of our study.

3.2.1 *Classification Tasks*

We address three classification tasks.

TRANS The first task, called *TRANS*, is to determine whether a given piece of JavaScript code has been transformed by any minification or obfuscation tool. We train classifiers for this task with examples of regular code, which has not been processed by any transformation tool, and with examples of transformed code, which has been processed by a minification or obfuscation tool.

OBFUS The second task, called *OBFUS*, is to determine whether a given piece of JavaScript code has been obfuscated. We train OBFUS classifiers with examples of regular code, examples of minified but not obfuscated code, and examples of obfuscated code.

TOOL-X The third task, called *TOOL-X*, is to determine for a given piece of transformed JavaScript code what tool has been used to transform the code. Because popular minification tools apply very similar transformations, we focus on obfuscation tools for this task.

3.2.2 *Training Data for Learning Classifiers*

To train the classifiers, we start with a set of human-written, or *regular*, JavaScript files and then create transformed variants of these files.

3.2.2.1 *Corpus of Regular Code*

The regular code examples are a subset of a corpus of 150,000 JavaScript files provided by others [Ray+16], which consists of human-written, non-transformed code from open-source projects. We remove from this corpus all files with a size less than 1kB, as they provide very little information for the classifiers to make an informed decision, and files with a size greater than 10kB, to keep the memory consumption during training at a manageable level.

3.2.2.2 *Program Transformation Tools*

Our study is based on popular minification and obfuscation tools. We select tools that are publicly available and widely used, as reported, e.g., by publicly visible download numbers.

MINIFICATION We consider seven minification tools (we use the names in parentheses as abbreviation throughout the chapter): Google Closure Compiler (closure), UglifyJS (uglify), babel-minify (babel), Matthias Mullie Minify (mmminify), javascript-minifier.com (jsmincom), and YUI Compressor (yui). These tools reduce the program size mainly by performing white spaces reduction and identifiers shortening. In addition, some tools, e.g., closure, also perform optimizations, such as inlining or constant folding.

OBFUSCATION We consider five obfuscation tools: javascript-obfuscator (jsobf), javascriptobfuscator.com (jsobfcom), jfogs, JSObfu, and DaftLogic Obfuscator (daft-logic). The result of our tool search also included JSFuck and javascript2img.com, but we exclude them as they either are unable to process large code files or produce invalid JavaScript code. Table 3.1 shows which transformation techniques the tools use, as reported in previous work and in the tool's documentation. The two most common obfuscation techniques are identifier encoding, usually using HEX encoding, and storing strings in a global array.

CONFIGURATIONS Most tools provide options to configure which transformation techniques to apply. Since different configurations may result in a different transformed code, we use multiple configurations for each tool. In total, we consider 15 configurations for the obfuscation tools and 31 configurations for the minification tools. Together with the regular version of a JavaScript file, this setup yields 47 variants of each file.

Transformation techniques	Obfuscation tools				
	jsobf	jsobfcom	jfogs	JSObfu	daft-logic
String splitting	✓				✓
Keyword substitution					
String concatenation				✓	
Encoding the entire code					✓
Encrypting the entire code					
Identifier encoding	✓	✓	✓	✓	✓
String encoding	✓	✓		✓	
Dead code injection	✓				
Control flow flattening	✓				
String array	✓	✓	✓		✓
Code protecting techniques	✓				

Table 3.1: Obfuscation tools and transformation techniques.

3.2.2.3 *Generation of Training Data*

As training data for a specific classifier, we randomly sample 10,000 files from the corpus of regular files and apply transformations tools to these files. For all three tasks, we train the classifiers with an even split of two classes of code, i.e., half of the training examples are expected to be classified as positive and negative, respectively. For the TRANS classifier, we apply the minification and obfuscation tools to each code example, using each tool equally often. For the OBFUS classifier, we obtain examples of obfuscated code by applying one of the obfuscation tools to each code example, using each tool equally often. To obtain examples of non-obfuscated code, we use regular and minified code. Since OBFUS gets trained to distinguish obfuscated code from both minified code and regular code, it can be used not only to identify obfuscated code among any code, but also to classify transformed code into minified versus obfuscated code. For the TOOL-X classifiers, we use examples of code transformed by tool X and code examples that are either not transformed or transformed by other tools.

3.2.3 *Classification via Identifier Frequencies*

The following describes the first of two approaches to learn classifiers. The approach exploits the fact that both minification and obfuscation tools replace the original identifiers, e.g., names of local variables, with other identifiers. Because many transformation tools use specific identifiers, a skilled human can determine whether a tool has been used and if yes, which tool. The learning approach is based on this observation.

The approach consists of two main steps. The first step is to extract a feature vector for a given JavaScript file. The feature vector summarizes what identifier names occur in the file and how frequent each name is. To this end, we tokenize the code and extract all identifiers. Based on all identifiers that occur in the training data, we determine a vocabulary of the 30,000 most common identifiers. Then, we transform each script into a feature vector of length 30,000, where each element represents a specific identifier. The element that represents a specific identifier is the *tf-idf* value of the identifier, i.e., the result of multiplying the *term frequency* with the *inverted document frequency* of an identifier. To compute the term frequency for a given script, we count the occurrences of each identifier and normalize these with the number of occurrences of the most frequent identifier in the script. For the inverted document frequency we divide, for each identifier, the total number of scripts in the training dataset by the number of scripts that contain the identifier and compute the logarithm of the resulting value.

The second step is to classify the feature vector of a JavaScript file using a support vector machine (SVM). We use SVMs because they are effective for binary classification problems. We achieve the best results for the SVM when using the *radial basis function kernel* and setting the penalty term C to 5. The classifier is implemented in Python using the machine learning library scikit-learn[1].

3.2.4 *Classification via AST Convolution*

The identifier-based classifier described above is conceptually simple but limited to a single feature of source code, i.e., identifier names. Our second classification approach addresses this limitation through a neural network that classifies abstract syntax trees (ASTs). ASTs are a useful representation of code because they preserve all relevant information while mak-

1 http://scikit-learn.org/stable

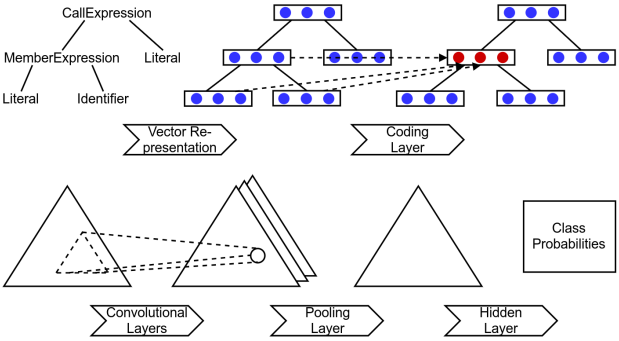


FIGURE 3.1: Tree-based convolutional neural network [Mou+16]. Rectangles with dots are vector representations of AST nodes; triangles represent trees.

ing the structural relationships between code elements explicit. To classify ASTs, we build upon a neural network architecture proposed by Mou et al. [Mou+16]. We adapt their approach by enriching traditional ASTs with additional information that proves useful for our classification tasks.

3.2.4.1 *Background: Tree-based Convolutional Neural Network*

We build upon an existing machine learning architecture for classifying trees, e.g., ASTs, based on a convolutional neural network. The network transforms a given tree into a continuous vector representation and then performs the actual classification task on the vector representation. The vector representation is learned in such a way that similar trees are represented by similar vectors. Figure 3.1 shows the five main steps involved in classifying trees. First, each node is transformed into a vector, where nodes with the same label, e.g., two CallExpression nodes, are mapped to the same vector. Second, a neural network layer ("coding layer") summarizes the children of a node and the node itself into the parent's vector. Third, several convolution layers extract features from the tree by sliding a feature detector over fixed-depth subtrees of the tree, which yields several trees of features. Fourth, a pooling layer summarizes these trees of features into a single tree again. Finally, all nodes of the new tree are passed through a fully connected hidden layer that outputs the classification result. During training with stochastic gradient descent, the parameters of the network are adapted to minimize a loss function that expresses how much the network's classification differs from the expected classification. Our imple-

mentation of the tree-based convolutional neural network is based on an implementation by Creston Bunch[2]. The architecture has previously been shown to be effective for identifying code that implements a particular algorithm [Mou+16]. We are the first to use it for identifying minified and obfuscated code.

3.2.4.2 *Enriched ASTs*

One possible approach is to apply the neural network to standard, out-of-the-box JavaScript ASTs. During initial experiments, we find this approach to provide a classifier with promising yet not fully satisfying accuracy. In particular, the classifiers struggle to learn from two features that are useful for a human but not well represented in standard ASTs: whitespace and specific properties of identifier names. Motivated by this observation, we enrich the standard JavaScript ASTs in two ways.

WHITESPACE The first enrichment is to add information about whitespace into ASTs. Usually, this information is abstracted away as it is irrelevant for most scenarios where ASTs are used. To recognize minified and obfuscated code, though, whitespace is relevant because both minification and obfuscation tools often remove all or at least some whitespace. We enrich ASTs with whitespace information as follows: Whenever two nodes in the AST correspond to successive elements in the source code, we check whether any whitespace exists between the two code elements. If no such whitespace exists, then we insert a new "no whitespace" node between the two adjacent nodes; otherwise, we leave the nodes unchanged.

LENGTH OF IDENTIFIERS The second enrichment of ASTs is to add information about the length of identifiers, i.e., the number of characters of an identifier name. The motivation is that several obfuscation tools replace identifiers with less understandable identifiers of a fixed length that differs from tool to tool. In contrast, regular code usually consists of natural identifiers that have variable lengths. We encode this information by modifying every "Identifier" AST node by appending the length of the identifier to the node label, e.g., "Identifier3" for an identifier `foo`. To deal with unusually long identifiers, any length exceeding 30 characters is simply represented by a single, special label.

2 https://github.com/crestonbunch/tbcnn

Classifier	Task		
	TRANS	OBFUS	TOOL-X
Identifier frequencies	76.40%	80.07%	64.44%–82.86%
AST convolution	95.06%	99.96%	99.68–100.00%

Table 3.2: Testing accuracies for different classifiers and classification tasks.

3.2.5 *Accuracy of Classifiers*

To decide which of the two classifiers to use for the study, we measure their classification accuracy on previously unseen sets of validation data. To this end, we randomly select 2,500 files from the non-transformed code (disjoint from the files used for generating training data), and create transformed variants of them, as described in Section 3.2.2.3. We then measure the accuracy for each classifier and task, i.e., the percentage of predictions that match the expected classification.

Table 3.2 summarizes the accuracy results for the two classifiers presented in Sections 3.2.3 and 3.2.4. Overall, the results show that both classifiers are effective (for comparison, a random decision would achieve 50% accuracy) and that the AST convolution-based classifier has the by far highest accuracy for all three tasks. In particular, the AST convolution-based classifier achieves more than 95% accuracy for all three tasks, and at least 99.68% accuracy for the OBFUS and TOOL-X tasks. In contrast, the classifier based on identifier frequencies performs poorly for some of the TOOL-X tasks, with an accuracy as low as 64.44% for one of the obfuscation tools.

To better understand why one classifier performs better than the other, we check how many of the different kinds of training examples the classifiers identify correctly. We find that the identifier frequency-based classifier is successful at identifying minified and obfuscated code but often fails to correctly identify regular code. For example, for the OBFUS task, the classifier correctly labels 99.92% of all obfuscated examples, but classifies only 59.47% of all non-obfuscated examples correctly. A detailed analysis of the identifiers that occur in obfuscated and non-obfuscated code explains these results: Many obfuscation tools use very characteristic identifiers, whereas regular code contains a wide range of natural identifiers. For example, jfogs creates many variables names `$fog$` followed by some number, and jsobf uses a similar pattern but also hex-encodes the identifiers.

To validate that enriching AST is beneficial over running the neural network on default JavaScript ASTs, we compare the accuracies of both variants of the AST-based classifier. We find that the default ASTs yield significantly lower accuracies than the enriched ASTs. For example, for the OBFUS task, the default ASTs give only 75.43% accuracy, which is not only much lower than with enriched ASTs but also lower than the classifier based on identifier frequencies.

Overall, we conclude from the accuracy results that the AST convolution-based classifier is highly effective at identifying transformed code, obfuscated code, and code obfuscated with a particular tool, making it a solid basis for studying JavaScript code at a large scale.

3.3 STUDYING DEPLOYED CLIENT-SIDE CODE

Based on the classifiers described above, this section presents the setup and results of our study of minification and obfuscation in deployed, client-side JavaScript code.

3.3.1 *Study Data: Deployed, Client-Side JavaScript Code*

To gather a representative set of JavaScript code used in real-world websites, we crawl the top 100,000 most popular websites, as listed by the Majestic Million[3] service. Our crawler visits each website, waits five seconds to enable dynamically loaded code to arrive, and then saves all scripts. We consider both code loaded via .js files and code loaded via inline scripts, i.e., via `<script>` tags without a `src` attribute. For the latter, the crawler copies the code between the tags into a new file. To speed up the loading of websites, we do not fetch resources other than scripts and HTML code.

The crawling yields 2,335,207 scripts from 85,001 websites in total. 14,999 websites are not accessible due to timeout errors and other reasons. Due to the limited size of ASTs that the classifiers can handle in reasonable time, we remove scripts with a size exceeding 40kB. This results in 1,861,489 scripts. We further remove scripts that are smaller than 512 bytes because we observe that such small scripts are hard to classify even for human subjects due to the limited number of clues that can aid the distinction between transformed and original scripts. Finally, we remove all those scripts for which the AST does not contain at least one "CallExpression" node. This is because such scripts are very often just configuration files in the JSON

3 https://de.majestic.com/reports/majestic-million

format. Applying these filters yields a set of 967,149 scripts. However, different websites may use the same JavaScript code, e.g., third-party libraries, thus the set of scripts contains duplicate files. In this study we are mostly concerned with the nature of the code on the web and not so much with its frequency on different websites, hence we remove all duplicates. This filtering leaves a final set of 424,023 unique JavaScript files, which we use for the study.

3.3.2 *Accuracy of Classifiers on Study Data*

Section 3.2.2.2 has established that our classifiers are highly accurate on code transformed by the tools used to generate the training data. To validate that the classifiers are effective also on the study data, for which we do not know which (if any) tools have been used, and to validate that the classification results match the classification that a skilled human could produce, we perform an experiment with JavaScript developers. The experiment involves five advanced developers, who have extensive experience in writing JavaScript and in understanding real-world JavaScript code.

The goal of the experiment is to gather ground truth to compare our classifiers against. During the experiment, each developer performs two tasks, which validate the TRANS and OBFUS classifiers, respectively. First, to validate the TRANS classifier, we show to each developer 50 scripts labeled by the classifier as transformed and 50 scripts labeled by the classifier as not transformed. We randomly sample the scripts from all 424,023 scripts in the study data. For each script, the developers are asked to answer the following question: "Is this human-written or generated/transformed code?" Second, to validate the OBFUS classifier, we repeat the same setup with 50 scripts labeled as obfuscated by the classifier and 50 scripts labeled as not obfuscated by the classifier. For these scripts, the developers are asked to answer the question: "Is this obfuscated or not obfuscated code? If the file is transformed but you are not sure if it is obfuscated, try to decide whether the developer transformed it for hindering understanding." To avoid any influence from file names, such as "jquery.min.js", we hide the original file names from the developers and assign some generic names to the files.

After aggregating the results of the experiment, we compute the inter-rater agreement, which quantifies the extent to which the participants agree with each other. More precisely, we compute Cohen's kappa for each pair of participants. The pairwise agreement ranges between 0.64 and 0.93,

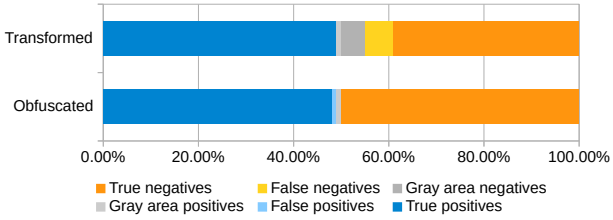


FIGURE 3.2: Effectiveness of the TRANS and OBFUS classifiers judged against the ground truth obtained from experienced JavaScript developers. The gray area depicts scripts for which the developers did not agree on the classification label.

with an average of 0.81, which is considered a very high agreement. To account for possible mistakes made by individual developers, we consider a 4-out-of-5 majority vote. That is, if at least four of the developers agree on whether a given script is transformed/obfuscated, then we consider this decision as the ground truth. If no such majority is reached, then we consider the scripts to be in a *gray area*, where even humans have a hard time judging whether the script has been transformed/obfuscated.

Figure 3.2 shows the results of the developer experiment. The blue area depicts true positives, i.e., scripts where the classifier and the developers agree on the scripts being transformed/obfuscated. Likewise, the orange area depicts true negatives, i.e., scripts where the classifier and the developers agree on the scripts being not transformed/obfuscated. Overall, the automated classifiers largely match the decisions by the developers. In addition, the TRANS classifier has 13% false negatives, i.e., it misses a few scripts that developers consider to be transformed, and the OBFUS classifier has 2% false positives, i.e., it sometimes incorrectly labels a non-obfuscated script as obfuscated. The gray parts in the middle of the figure, six scripts for TRANS and one for OBFUS, are the gray area, where the developers disagreed with each other.

Overall, we conclude from the experiment with developers that our automated classifiers match human classifications for the overwhelming majority of scripts. Given the 13% false negative rate of TRANS, one should interpret our results about the number of transformed scripts as a slight under-approximation of the actual number.

TRANS on all scripts:	Regular	61.5%
	Transformed	38.5%
OBFUS on all scripts:	Regular or minified	99.33%
	Obfuscated	0.67%
OBFUS on transformed scripts:	Minified	98.31%
	Obfuscated	1.69%

Table 3.3: Prevalence of transformed code.

3.3.3 *RQ1. Prevalence of Transformed Code*

To address the question of how prevalent minified and obfuscated code is in client-side JavaScript code, we classify all downloaded scripts using the TRANS and OBFUS classifiers. Table 3.3 summarizes the results. We find that 38.5% of all scripts have gone through some kind of transformation, including both minification and obfuscation. In contrast, only 0.67% of all scripts (2,842 scripts) have been obfuscated. Applying the OBFUS classifier to those scripts that are classified as transformed confirms the above numbers: The vast majority of transformed scripts are not obfuscated, i.e., they have only been minified. These numbers show that minification is popular in the web. This finding is in line with the fact that many JavaScript libraries and frameworks include a minification step in their deployment pipeline to reduce the file size and hence the transmission time. A possible explanation for the low number of obfuscated scripts is that obfuscation comes at a cost (discussed in detail in Section 3.3.7 and 3.3.8), and therefore it is used only when developers want to hide some behavior. Despite the surprisingly low number of obfuscated scripts, these scripts provide an interesting target for further analysis, and we provide them as a benchmark for future work.[4]

3.3.4 *RQ2. Prevalence of Obfuscation Tools*

Given the non-negligible number of obfuscated scripts, we next address the question which tools and techniques developers use for obfuscation. Table 3.4 shows for each tool X how many scripts are detected as obfuscated by this tool according to the corresponding TOOL-X classifier. The by far most popular obfuscator in the web is DaftLogic Obfuscator, with

4 http://software-lab.org/projects/obfuscation_study.html

Classifier	% detected scripts	% other scripts	# detected scripts
TOOL-JSObfu	0.01%	99.99%	3
TOOL-jsobfcom	0.04%	99.96%	149
TOOL-jfogs	0.02%	99.98%	0
TOOL-daft-logic	0.60%	99.40%	2,551
TOOL-jsobf	0.02%	99.98%	71

TABLE 3.4: Tools used to obfuscate scripts.

Pair of classifiers	Number of scripts
TOOL-jsobfcom ∩ OBFUS	147 (98% of TOOL-jsobfcom)
TOOL-daft-logic ∩ OBFUS	2,474 (97% of TOOL-daft-logic)
TOOL-jsobf ∩ OBFUS	71 (100% of TOOL-jsobf)

TABLE 3.5: Overlap between different classifiers.

2,551 scripts in total. As the only tool that encodes the entire code using the `eval` function, it clearly stands out among the other tools. The study data does not contain any scripts obfuscated with jfogs. A surprising fact from these results is that the apparent popularity on npm of tools like javascript-obfuscator and JSObfu does not transfer to client-side obfuscated code. One reason may be that these obfuscators are more popular for JavaScript code running on NODE.JS than for client-side code.

Having a set of classifiers related to obfuscation (OBFUS and several TOOL-X classifiers) raises the question to what extent the scripts detected by these classifiers overlap. Table 3.5 shows the overlap of scripts identified by the different obfuscation-related classifiers. We can make three observations. First, the TOOL-X classifiers do not overlap with each other, i.e., each of them precisely identifies scripts originating from a specific tool. This result is particularly remarkable for javascriptobfuscator.com and javascript-obfuscator, as these tools share a list of common obfuscation techniques. Second, almost all scripts detected as obfuscated by a specific tool are also detected as obfuscated by the general OBFUS classifier. Third, some scripts are classified as obfuscated but none of our TOOL-X classifiers can identify the tool used for obfuscating them.

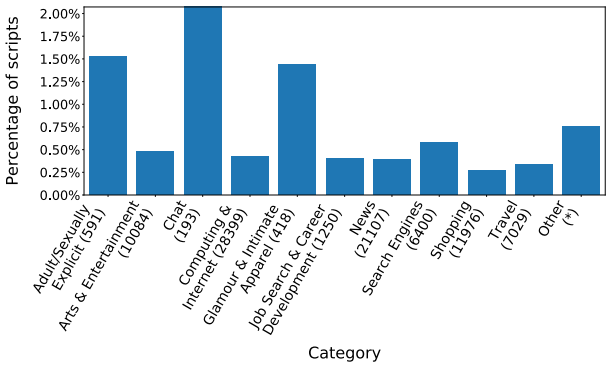


FIGURE 3.3: Prevalence of obfuscated code within categories of websites. In parenthesis on the x-axis we show the number of scripts in each category.

3.3.5 *RQ3. Transformations vs. Kinds of Scripts*

3.3.5.1 *Categories of Scripts*

We associate scripts with website categories, such as "News", "Travel", and "Education". To this end, we use the *Juniper Test-a-Site*[5] service, which yields a category for a given URL. We associate a category with a script based on the top-level domain from which the script was loaded.

OBFUSCATED CODE We first analyze the prevalence of obfuscated code loaded by websites in specific categories. Figure 3.3 shows for ten of the categories the percentage of obfuscated scripts among all scripts loaded by a site and an additional entry with the average across all other categories. The results show that the prevalence of obfuscation differs significantly across website categories. Categories with a particularly high percentage of obfuscated scripts include "Adult/Sexually Explicit" and "Glamour & Intimate Apparel", i.e., sites with content that careful users may trust less than an average website.

TRANSFORMED CODE Figure 3.4 shows the ratio between transformed and regular code within all script categories in the study data. The per-

5 http://mtas.surfcontrol.com/mtas/JuniperTest-a-Site.asp

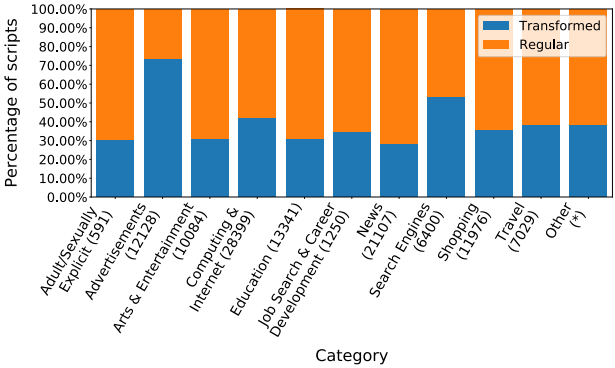


FIGURE 3.4: Prevalence of transformed code within categories of websites. In parenthesis on the x-axis we show the number of scripts in each category.

centage of transformed code ranges between 28% and 73%. The most significant categories are "Search engines" with 71.7% transformed code and "Advertisements" with 73.7%. We hypothesize that this is the case because websites in these categories serve scripts to many other domains, e.g. tracking scripts, and the size of the delivered scripts has a direct impact in the cost of using these services.

3.3.5.2 *Third-party Scripts*

The following studies whether transformations are particularly common for scripts loaded from third-party sites. As first-party scripts, we consider all inline scripts and all scripts loaded from the top-level domain of the visited website itself. All remaining scripts are considered third-party scripts.[6] Based on this grouping of scripts, we analyze how the results of the TRANS and OBFUS classifiers relate to where a script is loaded from.

We find that third-party scripts, with a percentage of 55.38%, are almost twice as frequently transformed than scripts loaded directly from the visited website, which have a percentage of only 30.18% transformed code. These findings seem natural because providers of third-party scripts, e.g., content-delivery networks, often provide a minified version to reduce load-

6 This classification may be wrong for websites that split their files across multiple top-level domains. Finding more accurate ways to identify third-party scripts is left for future work.

ing time. For scripts labeled as obfuscated, we do not find a significant difference between third-party and first-party scripts.

3.3.6 *RQ4. Runtime Behavior of Obfuscated Code*

To better understand what the obfuscated scripts actually do, we run them in a custom environment and observe what APIs they try to access. The custom environment consists of self-replicating proxy objects that emulate the browser APIs and other well-known frameworks required by the analyzed scripts. For example, we create a globally accessible `document` proxy object that returns another proxy each time one of its properties is accessed. This environment provides a simple and effective technique for inspecting the behavior of obfuscated scripts.

In total, we analyze 2,924 unique obfuscated scripts which include all scripts classified as obfuscated by either OBFUS or one of the TOOL-X classifiers. We use 13 self-replicating proxies to mock the browser API and we run our analysis in Node.js. After each run, we collect a trace summarizing the property writes, property reads, and function calls observed via the proxies. In total, we collect 3.6 million property accesses, 10,400 writes and 1.4 million function calls. 2,231 scripts access at least one property via the proxies and 1,263 scripts set globally accessible properties. These numbers show that our setup is effective at running the obfuscated scripts and at extracting meaningful information about the APIs they call and the properties they access.

Out of the 2,924 analyzed scripts, 341 access the `cookie` object, 316 the `userAgent`, 287 the `location` and 101 the `referrer`. These are all privacy-sensitive APIs that may be accessed to perform cookie theft, cookie syncing, referrer sniffing, or browser fingerprinting. However, a more detailed analysis is needed to confirm this hypothesis. The most frequently invoked method is by far `document.createElement`, used by 346 scripts. This means that in order to fully understand a given obfuscated script, an analysis needs to reason about the HTML code injected in the page, which seem to be a popular idiom. Moreover, we observe that 295 scripts call `document.createElement('script')`, i.e., inject code at runtime, a technique commonly used by malware.

We manually inspect some of the traces and find two security-relevant behavioral patterns. First, several traces contain various API calls known to be used for browser fingerprinting, such as `window.devicePixelRatio`, `navigator.plugins`, `screen.width` or `screen.colorDepth`. There

is even a trace in which after multiple such property accesses, a call to `document.createElement('img')` is made, which suggests that the obfuscated script is sending this information over the network. The second interesting case is of a trace containing a total of 336,000 invocations to `performance.now`, which is likely to be part of a timing attack.

An additional observation we make after our sampling-based manual analysis is that some scripts set global properties, such as `SHA256_init`, `jQueryPath`, `mtTracking` or `Fingerprint2`. Their names suggest that some of the scripts register benign functionality, such as SHA hash functions or jQuery, while others register tracking and fingerprinting functionality.

3.3.7 *RQ5. Performance of Transformed Code*

Code transformations may not only influence the understandability of code but also its efficiency. To better understand the cost-benefit trade-off of transformations, we study to what extent transformations affect the performance of code.

3.3.7.1 *Benchmarks*

Addressing RQ5 and RQ6 requires JavaScript code for which we can measure both the performance and the correctness. For this purpose, we gather a set of popular client-side JavaScript libraries that have extensive test suites. These tests include assertions to check the correctness of execution behavior, and they provide a reliable way to measure the execution time of code. Table 3.6 presents the ten libraries we consider, along with the number of tests they provide. All libraries are frequently used in client-side web applications. We execute their unit tests on Node.js, though, because it facilitates the performance measurements, as they are not influenced by opening and initializing a browser.

To study how transformations affect the performance and correctness of the libraries, we apply all 46 transformations (Section 3.2.2.2) to each library. To measure the performance of a given, possibly transformed, library, we execute its test suite 20 times and measure the overall wall-clock time of each execution. We do not perform a separate warm-up phase before measuring performance, as is common for longer-running benchmarks, to include the time for parsing code into our measurements, as this time affects the user experience on websites. Since closure injects code that is not compatible with our environment and prevents us from running the

Library	Category	Number of tests
Bacon	Reactive programming	7,492
async	Asynchronous programming	513
immutable	Immutable data structures	557
lodash	General utility	6,685
math	Math utility	4,063
moment	Date utility	3,232
ramda	Functional programming	954
underscore	General utility	1,569
voca	String utility	409
when	Promise implementation	872

Table 3.6: Libraries used to measure performance and correctness.

library tests, we omit this transformation tool for RQ5 and RQ6. We run all our performance and correctness measurements on an AMD Phenom II X6 1100T CPU with 16GB of RAM.

3.3.7.2 *Results*

Figure 3.5 shows the execution time of transformed code relatively to regular code for eleven different transformation tools. The figure presents results from the ten libraries listed in Section 3.3.7.1. Each data point shows the average execution time across 20 repetitions and the 95% confidence interval. The first six columns of data points are for minification tools, whereas the following four columns are for obfuscation tools. The last column shows the baseline, i.e. the performance of the original code.

For minification, the figure shows an overall improvement of efficiency. In particular, for two libraries, voca and math, minification causes performance improvements of over 20% and 8%, respectively. For most of the other libraries, minification causes a measurable but small performance improvement.

In contrast to minification, we observe an overall performance degradation after obfuscating code. For at least four libraries (ramda, moment, math, and Bacon), obfuscation significantly increases the execution time, with average increases between 16% to 37%. For eight out of the ten libraries, the obfuscated code is measurably slower than the regular code.

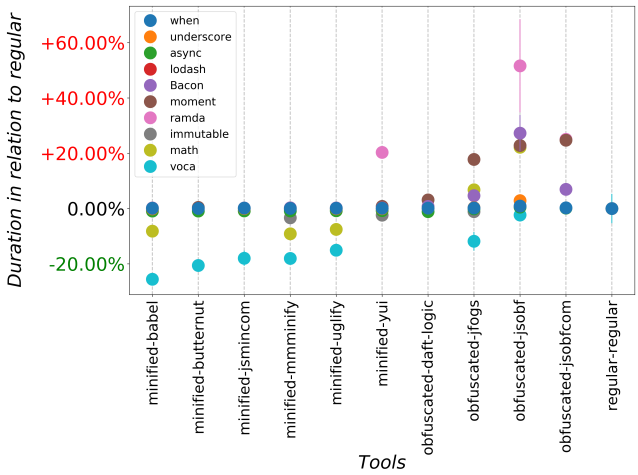


FIGURE 3.5: Execution time of transformed code relative to regular code. Each data point shows the average across 20 repetitions and the 95% confidence interval (which is too small to be visible for most libraries).

The only outlier is voca, which executes faster for two of the applied obfuscation tools.

3.3.8 *RQ6. Correctness of Transformed Code*

Besides affecting the performance of code, there is another potential cost of applying code transformations: the impact of transformations on the correctness of the code. To assess this impact, we run the test suites of the benchmarks from Section 3.3.7.1 before and after applying different transformations and measure the percentage of tests that still succeed after the transformation. We say that transformed code is *correct* if this percentage is 100%, i.e., the transformed code passes all tests.

Figure 3.6 shows the percentage of correct code among all code transformed with a specific tool. Overall, only about 70% of the minified code is correct, and even worse, less than 50% of the obfuscated code is correct. The original code, shown in the right-most column, is by definition 100% correct. A manual analysis of transformed code that fails test cases shows two root causes. First, some transformation tools have implementation-level bugs that get triggered by some code to be transformed. For ex-

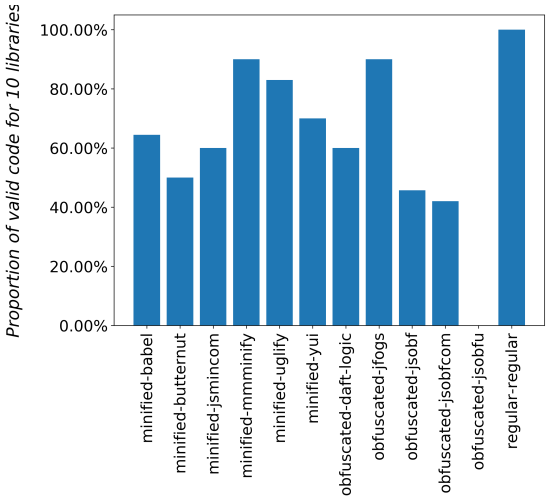


FIGURE 3.6: Percentage of correct code among all code transformed by a specific tool.

ample, JSObfu, which consistently creates invalid code, extensively uses the `String.fromCharCode()` method to encode constant string occurring in the code. However, in some cases the method is applied on an undefined object instead of a string, which causes an exception. Second, some transformation tools change the semantics of code in ways that affect rather subtle corner-cases of the JavaScript language. For example, some configurations of UglifyJS replace non-global function names with short and meaningless names. If such a function is a constructor function, this transformation affects code that creates an object with this function and then checks the name of the constructor using JavaScript's reflection APIs. Some of the tests trigger this corner case, e.g., a test of the "immutable" benchmark checks whether objects of type `Record` have a constructor with the same name.

We conclude from these results that transformation tools may not only impose a performance cost, but even worse, risk to change the semantics of code. A practical take-away from this finding is that users of such tools must carefully check the correctness of transformed code, instead of blindly relying on the transformation tool. Our results also motivates future work on validating and improving minification and obfuscation tools, e.g., through automated testing.

3.4 CONCLUSIONS

This chapter presents the first large-scale study of minified and obfuscated JavaScript in client-side website applications. Our study leads to several findings that should be of interest for both the web community and the security community. In particular, we show that transformed code is surprisingly common, whereas obfuscation aimed at hindering understanding remains an exception. Yet, there is a non-negligible number of obfuscated scripts, which typically occur in specific website categories and which sometimes expose security-relevant behavior, e.g., fingerprinting or timing attacks. Moreover, we show that obfuscation not only provides benefits to its users but also imposes a cost by negatively impacting both the performance and the correctness of the code. Besides these findings we are releasing the obfuscated code detected during the study to stimulate future research in this area.

Part II

Vulnerabilities and Attacks

4

INJECTION VULNERABILITIES ON THE SERVER-SIDE

In this chapter we present injection vulnerabilities in Node.js, a class of vulnerabilities that has more serious implications in the server-side than on the client-side due to the novel threat model (see particularity P_1 in the introduction). Once again, we concentrate on libraries, but our findings have direct implications for the security of full-stack web applications. This chapter shares material with the corresponding publication [SPL18].

4.1 MOTIVATION

As mentioned in the introduction, JavaScript has recently become increasingly popular for platforms beyond the browser: server-side and desktop applications that use Node.js, mobile programming (Apache Cordova/-PhoneGap); it is even used for writing operating systems, such as Firefox OS. One of the forces behind using JavaScript in other domains is to enable client-side programmers to reuse their skills in other environments.

Unfortunately, this skill transfer also spreads the risk of misusing the language in a way that threatens software security. One example of this is bad programming habits of client-side JavaScript, such as the widespread use of the `eval` construct [Ric+11], spreading to the emerging platforms. Additionally, new types of vulnerabilities and attacks become possible, which do not directly map to problems known in the client-side domain. For example, recent work shows that mobile applications written in JavaScript contain injection vulnerabilities [Jin+14] and that the impact of attacks in mobile applications is potentially more serious than that of client-side cross-site scripting (XSS). Others have shown how perilous the use of dangerous and outdated JavaScript APIs can be [Lau+17]. Companies like the Node Security Platform[1] and Snyk[2] are maintaining vulnerability data for platforms that include Node.js and Ruby Gems, underlining the importance of these issues, but do no provide actionable prevention strategies.

This chapter presents the first work to thoroughly investigate a security issue specific to JavaScript executed on the Node.js platform. Specifically,

1 https://nodesecurity.io/
2 https://snyk.io/

we focus on injection vulnerabilities, i.e., programming errors that enable an attacker to inject and execute malicious code in an unintended way. Injection vulnerabilities on the Node.js platform differ from those on other JavaScript platforms in three significant ways.

1) Injection APIs and impact of attacks: Node.js provides two families of APIs that may accidentally enable injections. The `eval` API and its variants take a string argument and interpret it as JavaScript code, allowing an attacker to execute arbitrary code in the context of the current application. The `exec` API and its variants take a string argument and interpret it as a shell command, giving the attacker access to system-level commands, beyond the context of the current application. Moreover, attackers may combine both APIs by injecting JavaScript code via `eval`, which then uses `exec` to execute shell commands. Because of these two APIs, and because Node.js lacks the security sandbox of the web browser, injection vulnerabilities in Node.js can cause significantly more harm than in the browser, e.g., by modifying the local file system or even taking over the entire machine.

2) Developer stance: While it is tempting for researchers to propose an analysis that identifies vulnerabilities as a solution, to have longer-range impact, it helps to understand Node.js security more holistically. By analyzing security issues reported in the past and through developer interactions, we observe that, while injection vulnerabilities are indeed an important problem, developers who use and maintain JavaScript libraries are generally reluctant to use analysis tools and are not always willing to fix their code.

To better understand the attitude of Node.js module developers toward potential injection flaws, we submitted a sample of 20 bug reports to developers on GitHub. Somewhat to our surprise, only about half the reports were attended to and only a small fraction was fixed (the results of this experiment are detailed in Figure 4.4). To understand the situation further, we reviewed many cases of the use of `eval` and `exec`, to discover that most (80%) could be easily refactored by hand, eliminating the risk of injections [Mea+12]. These observations suggest that even given the right analysis tool, it is *unlikely* that developers will proceed to voluntarily fix potential vulnerabilities.

3) Blame game: A dynamic we have seen developing is a *blame game* between Node.js module maintainers and developers who use these modules, where each party tries to claim that the other is responsible for checking untrusted input. Furthermore, while an individual developer can find

it tempting to deploy a *local fix* to a vulnerable module, this patch is likely to be made obsolete or will simply be overwritten by the next module update. These observations motivated us to develop an approach that provides a high level of security with a very small amount of developer involvement.

The main contribution of this chapter is to present a study of injection vulnerabilities in 235,850 Node.js modules, focusing on why and how developers use potentially dangerous APIs and whether developers appear open to using tools to avoid these APIs.

4.2 Background and Example

Injection APIs in Node.js: The Node.js platform provides two families of APIs that may allow an attacker to inject unexpected code, which we call *injection APIs*. First, `exec` enables command injections if an attacker can influence the string given to `exec`, because this string is interpreted as a shell command. The `exec` API has been introduced by Node.js and is not available in browsers. Second, calling `eval` enables code injections if an attacker can influence the string passed to `eval`, because this string is interpreted as JavaScript code. Since code injected via `eval` may contain calls to `exec`, any code injection vulnerability is also a command injection vulnerability. The latter distinguishes server-side JavaScript from the widely studied client-side problems of `eval` [Ric+11] and introduces an additional security threat. In this chapter, we focus on `exec` and `eval`, as these are the most prominent members of the two families of APIs. Extending both the study in this chapter and our mitigation mechanism in Chapter 7 to more APIs, e.g., `new Function()` or modules, e.g., `shelljs` is straightforward. Moreover, the approach can also be applied with minimal effort to other types of security vulnerabilities, e.g. SQL injections and path traversals.

In contrast to the browser platform, Node.js does not provide a security sandbox that controls how JavaScript code interacts with the underlying operating system. Instead, Node.js code has direct access to the file system, network resources, and any other operating system-level resources provided to processes. As a result, injections are among the most serious security threats on Node.js, as evidenced by the Node Security Platform[3], where, at the time of writing, 20 out of 66 published security advisories address injection vulnerabilities.

3 https://nodesecurity.io/advisories/

```
1  function backupFile(name, ext) {
2    var cmd = [];
3    cmd.push("cp");
4    cmd.push(name + "." + ext);
5    cmd.push("~/.localBackup/");
6
7    exec(cmd.join(" "));
8
9    var kind = (ext === "jpg") ? "pics" : "other";
10   console.log(eval("messages.backup_" + kind));
11 }
```

FIGURE 4.1: Motivating example.

Module system: As discussed in Chapter 2, code for NODE.JS is distributed and managed via the *npm* module system. A module typically relies on various other modules, which are automatically installed when installing the module. There is no mechanism built into npm to specify or check security properties of third-party modules before installation.

Motivating example: Figure 4.1 shows a motivating example that we use throughout this chapter to illustrate the problem and in Chapter 7 for presenting a defense mechanism against injections. The function receives two parameters from an unknown source and uses them to copy a file on the local file system. The parameters are intended to represent a file name and a file extension, respectively. To copy the file, lines 2 to 5 construct a string that is passed as a command to `exec` (line 7), which will execute a shell command. The code also logs a message to the console. Line 10 retrieves the content of the message by looking up a property of the `messages` object. The property and the message depend on the extension of the backed up file. Implementing a lookup of a dynamically computed property with `eval` is a well-known misuse of `eval` that frequently occurs in practice [Ric+11]. For example, suppose the function is called with `backupFile("f", "txt")`. In this case, the command will be `cp f.txt ~/.localBackup` and the logged message will be the message stored in `messages.backup_other`.

The example contains two calls to APIs that may allow for injecting code (lines 7 and 10). As an example for an injection attack, let us consider the following call:

```
backupFile("-help && rm -rf * && echo ", "")
```

The dynamically constructed command will be:

```
cp -help && rm -rf * && echo . ~/.localBackup/
```

Unfortunately, this command does not backup any files but instead it creates space for future backups by deleting all files in the current directory. Such severe consequences distinguish the problem of injections on Node.js from injections known from client-side JavaScript, such as XSS: because Node.js code runs without any sandbox that could prevent malicious code from accessing the underlying system, an attacker is able to inject arbitrary system-level commands.

4.3 A Study of Injection Vulnerabilities

To better understand how developers of JavaScript for Node.js handle the risk of injections, we conduct a comprehensive empirical study involving 235,850 npm modules. We investigate four research questions (RQs).

4.3.1 *RQ1: Prevalence of Calls to Injection APIs*

At first, we study whether APIs that are prone to injection vulnerabilities are widely used in practice. We call a module that directly calls an injection API an *injection module*. To assess whether a module uses another module that calls an injection API, we analyze dependences between modules, as specified in their `package.json` file. Given an injection module m_{inj}, we say that another module m_1 has a level-1 (level-2) dependence if it depends on m_{inj} (via another module). Figure 4.2 shows how many npm modules use injection APIs, either directly or via another module. We find that 7,686 modules and 9,111 modules use `exec` and `eval`, respectively, which corresponds to 3% and 4% of all modules. In total, 15,604 modules use at least one injection API. Furthermore, about 20% of all modules depend directly or indirectly on at least one injection API.

Estimating the potential effect of protecting vulnerable modules shows that fixing calls to the injection APIs in the most popular 5% of all injection modules will protect almost 90% of all directly dependent modules. While this result is encouraging, it is important to note that 5% of all modules still corresponds to around 780 modules, i.e., many more than would be reasonable to fix manually. Moreover, manually fixing these modules now would be a point-in-time solution that does not ensure the safety of future versions of modules and of new modules.

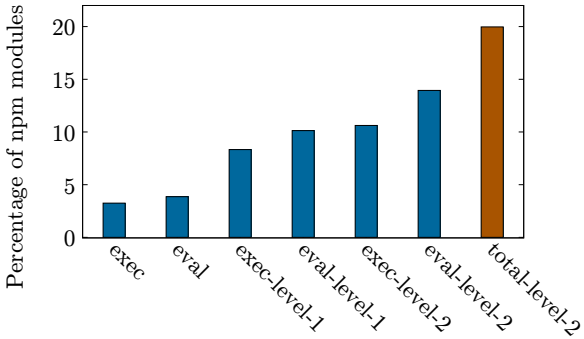


FIGURE 4.2: Prevalence of uses of injection APIs in npm modules.

4.3.2 *RQ2: Usage Patterns for Injection APIs*

To understand why developers use injection APIs, we identify recurring usage patterns and check whether the usages could be replaced with less vulnerable alternatives. We manually inspect a random sample of 50 uses of `exec` and 100 uses of `eval` and identify the following patterns.

Patterns of `exec` usage: The majority of calls (57%) trigger a single operating system command and pass a sequence of arguments to it. For these calls, the developers could easily switch to `spawn`, which is a safer API to use, equivalent to the well-known `execv` functions in C. The second-most common usage pattern (20%) involves multiple operating system commands combined using Unix-style pipes. For this pattern, we are not aware of a simple way to avoid the vulnerable `exec` call. The above-mentioned `spawn` accepts only one command, i.e., a developer would have to call it multiple times and emulate the shell's piping mechanism. Another interesting pattern is the execution of scripts using relative paths, which accounts for 10% of the analyzed cases. This pattern is frequently used as an ad-hoc parallelization mechanisms, by starting another instance of NODE.JS, and to interoperate with code written in other programming languages.

Patterns of `eval` usage: Our results of the usage of `eval` mostly match those reported in a study of client-side JavaScript code [Ric+11], showing that their findings extend to NODE.JS JavaScript code. One usage pattern that was not previously reported is to dynamically create complex functions. This pattern, which we call "higher-order functions", is widely used in server-side JavaScript for creating functions from both static strings and

```
function escape(s) {
  return s.replace(/"/g, '\\"');
}
exports.open = function open(target, callback) {
  exec(opener + ' "' + escape(target) + '"');
}

// Possible attack: open("`rm -rf *`");
```

FIGURE 4.3: Regular expression-based sanitization and input that bypasses it.

user-provided data. We are not aware of an existing technique to easily refactor this pattern into code that does not use `eval`.

Overall, we find that over 20% of all uses of injection APIs cannot be easily removed. Furthermore, many of the remaining uses are unlikely to be refactored by the developers, e.g., because techniques for removing usages of `eval` [JJM12; Mea+12] are available but not adopted by developers.

4.3.3 *RQ3: Existing Mitigation Against Injection Attacks*

To understand how developers deal with the risk of injections, we study to what extent data gets checked before being passed into injection APIs. Specifically, we analyze two conditions. First, whether a call site of an injection API may be reached by attacker-controlled data, i.e., whether any mitigation is required. We consider data as potentially attacker-controlled if it is passed as an input to the module, e.g., via a network request, or if the data is passed from another module and then propagates to the injection call site. Second, if the call site requires mitigation, we analyze which mitigation technique the developers use. We find that 58% of the inspected call sites are exploitable, i.e., attacker-controlled data may reach the injection API. Among these call sites, the following mitigation techniques are used:

None: A staggering 90% of the call sites do not use any mitigation technique at all. For example, the call to `exec` in the motivating example in Figure 4.1 falls into this category.

Regular expressions: For 9% of the call sites, the developers harden their module against injections using regular expression-based checks of input data. An example of such a fix in our data set is shown in Figure 4.3. Unfortunately, most regular expressions we inspected are not correctly implemented and cannot protect against all possible injection attacks. For exam-

ple, the `escape` method in Figure 4.3 does not remove back ticks, allowing an attacker to deliver a malicious payload using the command substitution syntax, as illustrated in the last line of Figure 4.3. In general, regular expressions are fraught with danger when used for sanitization [Hoo+11].

Sanitization modules: To our surprise, none of the modules uses a third-party sanitization module to prevent injections. To validate whether any such modules exists, we searched the npm repository and found six modules intended to protect calls to `exec` against command injections: shell-escape, escapeshellarg, command-join, shell-quote, bash, and any-shell-escape. In total, 198 other modules depend on one of these sanitization modules, i.e., only a small fraction of the 19,669 modules that directly or indirectly use `exec`. For `eval`, there is no standard solution for sanitization and the unanimous expert advice is to either not use it at all in combination with untrustworthy input, or to rely on well tested filters that allow only a restricted class of inputs, such as string literals or JSON data.

We conclude from these results that a large percentage of the 15,604 modules that use injection APIs are *potentially vulnerable*, and that standard sanitization techniques are rarely used. Developers are either unaware of the problem in the first place, unwilling to address it, or unable to properly apply existing solutions.

4.3.4 *RQ4: Maintainability of Vulnerable Npm Modules*

To understand whether module developers are willing to prevent vulnerabilities, we reported 20 previously unknown command injection vulnerabilities to the developers of the modules that call the injection APIs. For each vulnerability, we describe the problem and provide an example attack. Most of the developers acknowledge the problem. However, in the course of several months, only three of the 20 vulnerabilities have been completely fixed, confirming earlier observations about the difficulty of effectively notifying developers [Dou+11; Dur+14; Sto+16].

One may hypothesize that these vulnerabilities are characteristic to unpopular modules that are not expected to be well maintained. We checked this hypothesis by measuring the number of downloads between January 1 and February 17, 2016 for three sets of modules: (i) modules with vulnerabilities reported either by us or by others via the Node Security Platform, (ii) all modules that call an injection API, (iii) all modules in the npm repository. Figure 4.4 summarizes our results on a logarithmic scale. The boxes are drawn between the lower quartile (25%) and the upper one (75%) and

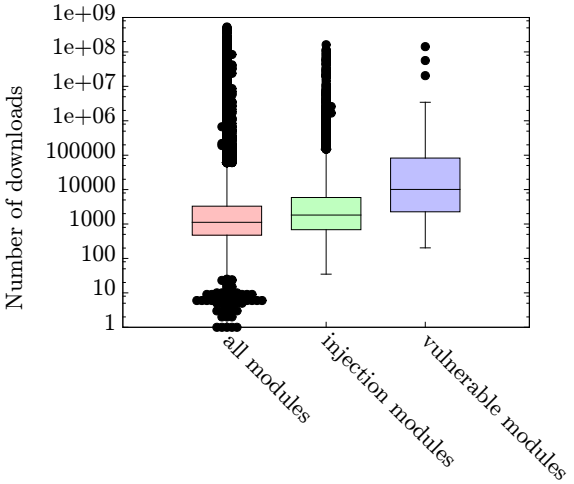


FIGURE 4.4: Comparison of the popularity of all the modules, modules with calls to injection APIs, and modules with reported vulnerabilities. The boxes indicate the lower quartile (25%) and the upper quartile (75%); the horizontal line marks the median; the dots are outliers.

the horizontal line marks the median. The results invalidate the hypothesis that vulnerable modules are unpopular. On the contrary, we observe that various vulnerable modules and injection modules are highly popular, exposing millions of users to the risk of injections.

4.3.5 *Case Study: The growl Module*

To better understand whether developers are aware of possible injection vulnerabilities in modules that they use, we manually analyzed 100 modules that depend on growl. The growl module displays notifications to users by invoking a particular command via `exec`, which is one of the vulnerabilities we reported as part of RQ4. We found that modules depending on growl pass various kinds of data to growl, including error messages and data extracted from web pages. As anticipated in RQ1, vulnerabilities propagate along module dependences. For example, the loggy module exposes the command injection vulnerability in growl to 15 other modules that de-

```
// sanitization in autolint
function escape(text) {
  return text.replace('$', '\\$');
}
// sanitization in mqtt-growl
message = message.replace(/"/g, "\\\"");
// sanitization in bungle
const ansiRx =
  /[\u001b\u009b][[()#;?]*
  (?:[0-9]{1,4}(?:;[0-9]{0,4})*)?
  [0-9A-ORZcf-nqry=><]/g;
Growl(message.replace(ansiRx, ''));
// sanitization in chook-growl-reporter
function escapeForGrowl(text) {
  var escaped = text.replace(/\(/g, '\\(');
  escaped = escaped.replace(/\)/g, '\\)');
  escaped = escaped.replace(/\""/g, '\\"');
  return escaped;
}
// input that bypasses all the sanitizations:
// "tst`rm -rf *`";
```

FIGURE 4.5: Broken sanitization in growl's clients.

pend on loggy by sending inputs directly to growl without any check or sanitization.

We found only four modules that sanitize the data before sending it to the vulnerable module: autolint, mqtt-growl, bungle, and chook-growl-reporter. We report these sanitizers in Figure 4.5. Sadly, we find that all these methods are insufficient: one can easily bypass them, as illustrated by the example input at the end of Figure 4.5. The input again exploits the command substitution syntax, which is not considered by any of the sanitizers.

Impact of our study: After we published a preliminary version of this paper [SPL16], several providers of NODE.JS vulnerability databases included findings of the study as vulnerability reports.[4]

4.4 CONCLUSIONS

This chapter studies in detail injection vulnerabilities in NODE.JS and shows that the problem is widespread and not yet adequately addressed. Moreover, we show that the developers are slow to fix the reported vulnerabilities in their library and that they tend to delegate sanitization responsibility to the consumers of their code.

4 https://snyk.io/vuln/page/2?type=npm, CWE-94, npm:nd-validator:20160408 and https://nodesecurity.io/advisories.

5

REDOS VULNERABILITIES ON THE SERVER-SIDE

In this chapter we present another class of security vulnerabilities that are made worse by the novel threat model for JavaScript (see particularity P_1 in the introduction): Regular expression denial of service (ReDoS), a type of algorithmic complexity attack. We also present a novel methodology that enables an adversary to leverage vulnerabilities in open-source libraries for attacking full-stack JavaScript web applications. This chapter shares material with the corresponding publication [SP18].

5.1 MOTIVATION

Regular expressions are widely used in all kinds of software. Since regular expressions are easy to get wrong [Wil04], which may help attackers to bypass checks [BBJ10; Hoo+11], developers are trained to think about the correctness of regular expressions. In contrast, another security-related aspect of regular expressions is often neglected: the performance, specifically, how long it takes to match a string against a regular expression. Unfortunately, given a specifically crafted input, matching against a suboptimally designed regular expression can easily take several minutes or even hours. For example, matching the apparently harmless regular expression `/(a+)+b/` against a sequence of 30 "a" characters on the Node.js JavaScript platform takes about 15 seconds on a standard computer.[1] Matching a sequence of 35 "a" characters already takes over 8 minutes, i.e., the matching time explodes exponentially.

If a server implementation suffers from this kind of performance problem, then an attacker can exploit it to overwhelm the server with hard-to-match inputs. This attack is known as *regular expression denial of service*, or short *ReDoS*. Such attacks are a form of algorithmic complexity attack [CW03] that exploits the worst-case complexity behavior of algorithms that match a string against a regular expression. Since for some regular expressions, the worst-case complexity is much higher than the

1 We use JavaScript syntax for regular expressions, i.e., a pattern is either enclosed by slashes or given to the `RegExp()` constructor.

average-case complexity, an attacker can cause denial of service with a few, relatively small inputs.

Even though ReDoS has been known for several years, recent developments in the web server landscape bring new and increased attention to the problem. The reason is that, as mentioned several times so far in this dissertation, JavaScript is becoming increasingly popular not only for the client-side but also for the server-side of web applications. However, the single-threaded nature of JavaScript, where every request is handled by the same thread, makes server applications much more susceptible to ReDoS attacks. In practice, to avoid making the server unresponsive by blocking this thread, developers try to split any long-running computation into smaller events, which are than handled asynchronously. The problem is that in current JavaScript engines, matching a string against a regular expression cannot be easily split into multiple chunks of computation. As a result, a single request can effectively block the main thread, making the web server unresponsive to any other incoming requests and preventing it from finishing any other already established requests.

Despite the importance of ReDoS in web servers, there is currently little reported knowledge about the prevalence of ReDoS vulnerabilities in real-world websites. In this chapter, we present the first comprehensive study of ReDoS across a large number of websites. We seek to answer the following questions:

- How widespread are ReDoS vulnerabilities in the server-side part of real-world JavaScript-based websites?
- What is the effect of vulnerabilities on the response time of web servers?
- What kinds of vulnerabilities are the most prevalent?
- Are more popular websites less vulnerable to ReDoS?
- Are existing defense mechanisms in use and if so, how effective are they in preventing ReDoS attacks?

Answering these questions involves solving two methodological challenges. First, how to identify ReDoS vulnerabilities in the server-side of websites when their source code is not available. We address this challenge based on a set of 25 previously unknown vulnerabilities in popular libraries and by speculating how these libraries may be used in servers. Second, how to analyze which websites are exploitable without actually

performing a denial of service attack against live websites. We address this challenge by triggering requests with increasing input size, using both manually crafted exploit inputs and randomly generated, harmless inputs, and by statistically comparing the response times.

Using this methodology, we identify 339 websites that suffer from at least one ReDoS vulnerability. Based on experiments with locally installed versions of the vulnerable server-side libraries, attacking these websites with crafted inputs can cause a web server to remain unresponsive for several seconds or even minutes. These problems are due to a very small number of vulnerabilities, with a single vulnerability that causes 241 sites to be exploitable. While this is encouraging from a mitigation point of view, it also implies that an attacker aware of a single, previously unknown vulnerability can cause serious harm to several websites.

Ojamaa and Düüna [OD12] were the first to identify ReDoS as a threat for the Node.js platform. Davis et al. [DKL17] confirm that such problems exist in popular modules and report that 5% of the security vulnerabilities identified in Node.js libraries are ReDoS. No prior work has studied the impact of ReDoS on real-world web sites. Existing work on detecting ReDoS vulnerabilities mostly targets languages other than JavaScript. For example, Wüstholz et al. [Wüs+17] propose a static analysis of ReDoS vulnerabilities in Java. The only available tool for JavaScript that we are aware of is a small utility called `safe-regex`[2], which checks for simple AST-level patterns known to cause ReDoS. However, this approach is notoriously prone to both false positives and false negatives, since it reasons neither about the context in which these patterns appear nor about the actual performance of regular expression matching. Our work shows the urgent need for effective tools and techniques that detect and prevent ReDoS vulnerabilities in JavaScript.

In summary, this chapter contributes the following:

- A novel methodology for analyzing the exploitability of deployed servers. The key ideas are (i) to hypothesize how server implementations may use libraries that have previously unknown vulnerabilities and (ii) to assess whether an attack is feasible without actually attacking the servers.
- The first comprehensive study of ReDoS vulnerabilities in JavaScript-based web servers. Out of 2,846 studied websites, we find 12% to be vulnerable.

2 https://www.npmjs.com/package/safe-regex

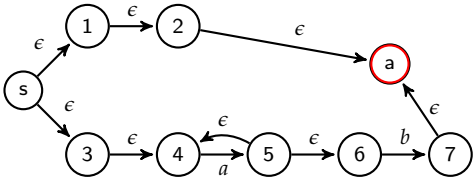


FIGURE 5.1: Automaton for the regular expression `/^(a+b)?$/`. s is the starting state and a is the accepting state.

- Empirical evidence that ReDoS is a real and widespread threat. Our work calls for novel tools and techniques that detect and prevent ReDoS vulnerabilities.
- A benchmark of previously unreported ReDoS vulnerabilities and ready-to-use exploits, which we make available for future research on finding, fixing, and mitigating ReDoS vulnerabilities:

  https://github.com/sola-da/ReDoS-vulnerabilities

5.2 BACKGROUND

5.2.1 *Regular Expression Matching*

Regular expressions are used to check whether a given sequence of characters *matches* a specified pattern. Most implementations in modern programming languages address this problem by converting the regular expression into an automaton [Tho68] and through a backtracking-based search for a sequence of transitions from the initial to an accepting state that consumes the given string. For example, consider the regular expression `/^(a+b)?$/` and its equivalent automaton in Figure 5.1. Given the string "aab", the automaton starts from state s and has two available transitions, to states 1 and 3. It first takes the transition to state 1, which leads to the accepting state a. Since the input string was not consumed and there are no available transitions, the algorithm backtracks to s and explores the transition to state 3 etc. After multiple explorations the algorithm identifies the sequence of transitions $s \to 3 \to 4 \to 5 \to 4 \to 5 \to 6 \to 7 \to a$, which reaches the accepting state and consumes all characters of the input string.

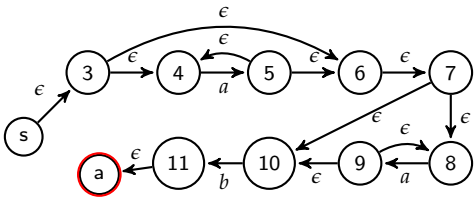


FIGURE 5.2: Automaton for the regular expression `/^a*a*b$/`. s is the starting state and a is the accepting state.

5.2.2 *Regular Expression Denial of Service (ReDoS)*

The backtracking-based search may cause the algorithm to backtrack possibly large number of times. ReDoS attacks exploit these pathological cases. For example, consider the regular expression `/^a*a*b$/`, its automaton in Figure 5.2, and the input string "aaa". Each character "a" can be matched using two transitions, $4 \rightarrow 5$ and $8 \rightarrow 9$. At each step, the algorithm needs to decide which of these two transitions to take. Eventually, since there is no character "b" in the input string, the algorithm will always fail when reaching state 11. However, before concluding that the input string does not match the pattern, the algorithm tries all possible ways of matching the "a" characters. The example is a regular expression of super-linear complexity [Wüs+17], since the number of transitions during matching is quadratic in the input size. Other regular expression even have exponential complexity, e.g., because of nested repetitions, such as in `/^(a*)*b$/`. In our study, we identify ReDoS vulnerabilities of both these types and show that both are of importance for server-side JavaScript.

5.2.3 *Execution Model of Server-Side JavaScript*

The server-side NODE.JS platform advocates a single-threaded, event-based execution model that uses asynchronous I/O calls. In NODE.JS, the main thread of execution runs an event loop, called the *main loop* that handles events triggered by network requests, I/O operations, timers, etc. A slow computation, e.g., matching a string against a regular expression, slows down all other incoming requests. Compared to multi-threaded web servers, such as Apache, the single-threaded execution model compounds the problem in JavaScript. For example, consider a regular expression that takes more than an hour to match, which we show to exist in widely used

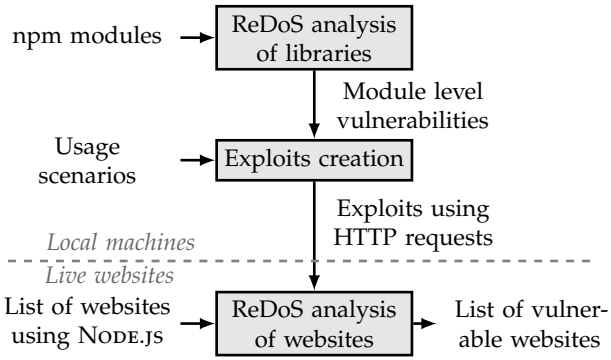


Figure 5.3: Overview of the methodology.

JavaScript software. To completely block an Apache web server, we need to send hundreds of such requests, each blocking one thread. Depending on the number of available parallel processing units, the operating system, and the thread pool size, new requests can still be handled even with hundred of busy threads running. In contrast, in Node.js one such request is enough to completely block the server for an hour. To make matters worse, even less severe ReDoS payloads can significantly degrade the availability of a Node.js server, as we show in Section 5.4.3.

5.3 Methodology

This section presents our methodology for studying ReDoS vulnerabilities in real websites. The overall goals of the methodology are to understand (i) how widespread such vulnerabilities are, (ii) whether an attacker could exploit them to affect the availability of live websites, and (iii) to what extent existing defense mechanisms address the problem. To answer these questions, our methodology must address two major challenges. The first challenge is a technical problem: Since the server-side source code of most websites is not available, how to know what vulnerabilities a website suffers from? The second challenge is an ethical concern: How to study the potential impact of attacks on live websites without actually causing noticeable harm to these websites?

Figure 5.3 shows a high-level overview of the methodology. We address the two challenges through experiments performed on machines under our control and on live websites. The main insight to address the first challenge

is to use previously unknown vulnerabilities in popular JavaScript libraries and to speculate how servers may use these libraries. More precisely, we analyze third-party libraries, called node package manager modules (npm packages or npm modules for short, discussed in detail in Chapter 2), to find vulnerabilities that may be exploitable via HTTP requests. We then hypothesize how the server implementation may use these packages and create exploits for these scenarios.

To address the second challenge, we present a technique that tests if a site is vulnerable but that avoids blocking the site for a noticeable amount of time. The basic idea is to start with very small payloads that do not require more computation time than normal web requests, and to then slowly increase the payload – just long enough to claim with confidence that the site *could* be exploited if an attacker used larger payloads. To decide on the size of payloads sent to live websites, we run experiments on locally installed web servers that use the vulnerable packages.

An alternative to experimenting with live websites would be to locally install open-source web applications. We discarded this idea because it would limit the scale of our study to the few web sites that disclose their server-side code, because it would remain unclear whether the results generalize to real-world sites, and because we could not study which countermeasures are deployed in practice.

5.3.1 *Identifying Websites with Server-Side JavaScript*

We consider the most popular one million websites aggregated by Alexa[3] as candidates for our study. Many of these websites do not use JavaScript on the server-side and analyzing all the websites against our exploits is prohibitive. Instead, we select sites that run the currently most popular framework for JavaScript-based web servers, Express[4]. To this end, we make a request to each of the one million websites and check whether the header `X-Powered-By` is "Express". The framework sets this value by default on a fresh installation. In total, 2,846 sites set this header which account for a market share of around 0.3%, consistent with estimates by others.[5] Because headers may be filtered to prevent attackers from targeted attacks and because frameworks other than Express exist, our selection of sites is likely yield an underapproximation of the impact of ReDoS. Figure 5.4 shows the

3 http://www.alexa.com/
4 https://expressjs.com/
5 https://w3techs.com/technologies/details/ws-nodejs/all/all

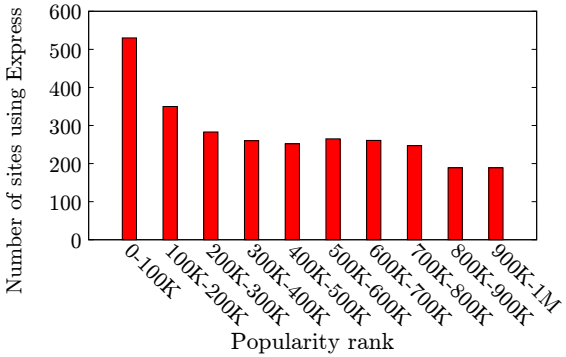


FIGURE 5.4: Number of server-side JavaScript websites within a given popularity range.

number of Express-based websites in batches of 100,000 sites, ordered by popularity. We observe that Express tends to be used by the more popular websites, confirming the importance of studying the security of JavaScript-based servers.

5.3.2 *Finding ReDoS Vulnerabilities in Libraries*

Our methodology relies on knowing previously unknown, or at least not yet fixed, ReDoS vulnerabilities in popular npm modules. Similar to previous work [Wüs+17], we consider a regular expression to be vulnerable if we can construct inputs of linearly increasing size that cause the matching time of the expression to increase super-linearly. To identify previously unknown vulnerabilities, we use a combination of automated and manual analysis, similar to what a potential attacker might do. This technique is not the contribution of this dissertation, but rather a way to enable our study. In principle, any other way of identifying ReDoS vulnerabilities could be used instead, including existing analyses [Wüs+17], which however, are currently not available for JavaScript.

At first, we download the 10,000 most popular modules and extract their regular expressions by traversing the abstract syntax trees of the JavaScript code. This yields a total of 324,791 regular expressions, with a mean of 63.67, a median of 5.00 and a maximum of 19,791 per module. After removing regular expressions that contain no repetitions, and hence are immune

to algorithmic complexity attacks, we obtain a total of 138,123 expressions, with mean 37.93 and median 4.00 per module.

Next, we semi-automatically search for regular expression patterns that are known to be vulnerable. For example, we search for expressions containing repetitions of a negated group followed by a character. The second regular expression in Figure 5.7 is an example because it contains the subexpression `[^=]+=`. A regular expression that is not anchored with a start anchor and contains this pattern is likely to be vulnerable. The reason is that the repetition group is generic enough to contain most of the possible prefixes and the = character guarantees that there exists a failing suffix. For example, the regular expression `/ab[^=]+=/` can be exploited using a long string `"abababab.."`.

Given a set of possibly exploitable regular expression, we manually inspect the context in which the regular expressions are used. The goal is to find matching operations on data that may be delivered through an HTTP request to a web server. To this end, we focus on (i) modules included in the Express framework, (ii) middleware modules that extend this framework, and (iii) modules that manipulate HTTP request components, such as the body or a specific header. For regular expressions in these modules, we keep only those with a possible data flow from the package interface or from an HTTP header to the regular expression. Overall, it took the author of this dissertation only a couple of days to find 25 such vulnerabilities in widely used npm modules, showing that a motivated individual can attack real-world websites with moderate effort. A more powerful attacker could easily detect a larger number of vulnerabilities and perform a larger-scale attack.

5.3.3 *Creating Exploits*

Based on the ReDoS vulnerabilities in npm modules, we create exploits targeted at web servers that use these modules. The main idea is to hypothesize how a server-side web application might use a module. To this end, we set up a fresh Express installation and implement an example web application that uses the module. For example, for a package that parses the user agent, we build an application that parses the user agent of every HTTP request for the main page, which might be used to track visitors. Next, we try to create an HTTP request where user-controlled data reaches the vulnerable regular expression, and craft input values that trigger an unusually long matching time. For crafting the input, we try to confuse

the regular expression engine by forcing it to backtrack because the input can be matched in multiple ways [KRT13; Wüs+17]. While creating exploits, we assume that the maximum header size is 81,750 characters, which is the default in Express.js. If we succeed in crafting an input that takes more than five seconds, we consider the vulnerability as exploitable and consider it for the remainder of the study.

To further assess the impact of the exploits, we measure how much longer it takes to process a crafted input compared to a random string of the same length. We use two ways of measuring the time. First, we measure the *matching time* of the regular expression, i.e., the time needed to check whether a string matches the regular expression. Second, we measure the time of an entire HTTP request, called *response time*. The response time may include various other components, such as HTTP parsing and serialization, DNS resolving, routing time for the package, and dealing with HTTP retransmissions or package fragmentation. To measure the response time of a site, we request its main page. For complex sites, this measure underapproximates the time a human user needs to wait for the page to load, because complex sites require separate requests for images, etc.

5.3.4 *ReDoS Analysis of Websites*

The next step is to measure how many websites are vulnerable to a ReDoS attack based on one of the exploits. The main challenge is to draw meaningful conclusions about the harm that an attacker *could* cause, without actually attacking live websites. During our initial experiments we sent one request with a crafted header that appeared to make the analyzed website unresponsive for almost a minute. The goal of our methodology is to avoid this type of mistake.

We address this challenge by triggering requests with increasing input sizes, using both crafted and random inputs, while measuring the response times. Based on locally performed experiments, we choose input sizes that are unlikely to block the server for more than a small, configurable amount of time (we use two seconds in our experiments). If the response time with crafted inputs grows faster than with random inputs, then we classify the website as exploitable.

Measuring the response time in a reliable way is non-trivial due to DNS resolving, network caching, delays, retransmissions, and other influencing factors. Another issue is how to determine whether the response time is larger than another in a statistically reliable way. We address these issues

by adapting a technique originally used for comparing the performance of software running on a virtual machine [GBE07; PHG14]. The basic idea is to repeatedly measure the response time and to conclude that crafted inputs cause a higher response time than random inputs only if we observe a statistically significant difference.

More specifically, to measure the response time for a given input, we first repeat the request n_w times to "warm up" the connection, e.g., to fill network caches, and then repeat the request another n_m times while recording the response times. Given k pairs of increasingly large random and crafted inputs $(i_{random}, i_{crafted})$, where the two inputs in a pair have the same size, we obtain k pairs (T_{random} and $T_{crafted}$) of sets of time measurements (with $|T_{random}| = |T_{crafted}| = n_m$). For each input size, we compare the confidence intervals of the values in T_{random} and $T_{crafted}$ and conclude that the response times differ if and only if the intervals do not overlap. If the response times differ for all k input sizes, we quantify the difference for an input size as the difference between $\overline{T}_{random}$ and $\overline{T}_{crafted}$, where $\overline{T}$ is the average of the times in T. For k input sizes, this comparison gives a sequence of differences $d_1, .., d_k$. Finally, we consider a website to be *exploitable* if $d_1 < d_2 < .. < d_k$. Intuitively, this means that the response times for random and crafted inputs have a statistically significant difference, and that this difference increases when the input size increases.

To execute these measurements, we need to pick values for n_w, n_m, k, and the k input sizes. We use n_w=three, n_m=five, and $k = 5$ because these values are large enough to draw statistically relevant conclusions for most websites yet small enough to not disturb the analyzed servers. For picking the k input sizes, the challenge is to ensure that we measure a difference when there is one without repeatedly causing the server to block for a longer period of time. We address this challenge by experimenting on a locally installed version of the vulnerable package and by choosing inputs that take approximately 100ms, 200ms, 500ms, 1s and 2s to respond to.

Our setup allows us to assess whether a website could be exploited without actually attacking it. Since we take measurements in a sequential manner and since the overall number of requests per site is small, we allow legitimate users to be served between our requests. Moreover, the servers of popular websites implement some kind of redundancy, such as multiple Node.js instances in a cluster, i.e., our measurements are likely to block only one such instance at a time. In contrast, an attacker would likely send both more requests and requests with larger inputs, which can cause severe harm to vulnerable sites, as we show in Section 5.4.3.

5.3.5 *Analysis of Mitigation Techniques*

Some sites reject requests with large headers and instead return a "400 Bad Request" error. This mitigation can limit the damage of ReDoS attacks. To measure whether a site uses this mitigation technique, we create benign requests of different sizes and measure how often a site rejects a request.

5.4 RESULTS

This section presents the results of applying the methodology described in Section 5.3 to live, real websites. We perform our measurements using three different machines depending on the experiments: a ThinkPad 440s laptop with four Intel i7 CPUs and 12GB memory (Section 5.4.1), a third party commercial web server with 512MB memory (Section 5.4.3 and 5.4.4) and a server with 48 Intel Xeon CPUs and 64GB memory (from Section 5.4.6 on).

5.4.1 *Vulnerabilities and Exploits*

Figure 5.5 shows the modules for which we found at least one vulnerable regular expression that can be exploited through the module's interface. At the time of performing our experiments, each vulnerability was working on the latest release of the package. The packages vary in the number of dependencies and downloads, but we can safely conclude that ReDoS vulnerabilities are present even in very popular packages.

Given the amount of possible damage entailed by the vulnerabilities, we have invested significant efforts to disclose them in a responsible way. For each vulnerability, we have contacted the developers either directly or through the Node Security Platform[6], and gave them several months to fix the problem before making it public. 14 of the 25 have been fixed by the time of writing and are listed as advisories on the Node Security Platform. For the others, the developers are either still in the process of fixing or decided to leave the task of fixing to the community. The complete list of vulnerabilities, along with details on their current status is available online.[7]

6 https://nodesecurity.io/advisories
7 https://docs.google.com/spreadsheets/d/1rnR8zsXeA1eccrpxeZK0_LtQOlc8j_u60IR7nnVQgbE/edit?usp=sharing

Module	Version	Number of dependencies	Downloads in July 2017
debug	2.6.8	16,055	54,885,335
lodash	4.17.4	49,305	44,147,504
mime	1.3.6	2,798	22,314,018
ajv	5.2.2	758	17,542,357
tough-cookie	2.3.2	302	15,981,922
fresh	0.5.0	197	14,151,270
moment	2.18.1	14,421	10,102,601
forwarded	0.1.0	31	9,883,630
underscore.string	3.3.4	2,486	7,277,966
ua-parser-js	0.7.14	225	5,332,979
parsejson	0.0.3	19	4,897,928
useragent	2.2.1	191	3,515,292
no-case	2.3.1	18	3,321,043
marked	0.3.6	2,624	3,012,792
content-type-parser	1.0.1	8	2,337,147
platform	1.3.4	128	757,174
timespan	2.3.0	34	523,290
string	3.3.3	911	421,700
content	3.0.5	9	316,083
slug	0.9.1	499	151,004
htmlparser	1.7.7	178	138,563
charset	1.0.0	36	112,001
mobile-detect	1.3.6	101	107,672
ismobilejs	0.4.1	50	44,246
dns-sync	0.1.3	7	10,599

FIGURE 5.5: Modules with at least one previously unknown vulnerability.

As explained in Section 5.3.3, we try to create exploits for the vulnerabilities by hypothesizing how web server implementations may use the vulnerable modules. Figure 5.6 shows the modules and usage scenarios for which we could create an exploit. For all the scenarios we assume the payload is sent using a specific HTTP header. We believe that HTTP bodies, UDP packages or WebSocket messages can also be used for the same purpose. The last column of Figure 5.6 shows the JavaScript implementation of the usage scenario. We run this implementation on our local server to experiment with the exploit.

Most of the scenarios and their implementations are relatively simple. This simplicity shows that an attacker that follows a methodology similar

ID	Module	Usage scenario	JavaScript example
1	charset	The website uses this package to parse the content type of every request.	`require("charset")(req.headers);`

2	content	The website uses this package to parse the content type of every request.	`var content = require("content");`
			`content.type(`
			`  req.headers["content-type"]);`

3	fresh	The website uses `express`, which by default uses this package to check the freshness of every request.	`var fresh = require("fresh");` `fresh(req.headers);`
4	forwarded	The website uses `express` and the "trust proxy" option is set. This package is then used to check which proxies a request came through.	`var forwarded = require("forwarded");` `var addrs = forwarded(req);`

5	mobile-detect	The website uses this package to get information about the requester.	`var MobileDetect =`
			`  require("mobile-detect");`
			`var headers =`
			`  req.headers["user-agent"];`
			`var md = new MobileDetect(headers);`
			`md.phone();`

6	platform	The website uses this package to get information about the requester.	`var platform = require("platform");`
			`var headers =`
			`  req.headers["user-agent"];`
			`var agent = platform.parse(headers);`

7	ua-parser-js	The website uses this package to get information about the requester.	`var useragent =`
			`  require("ua-parser-js");`
			`var headers =`
			`  req.headers["user-agent"]`
			`var agent = useragent.parse(headers);`

8	useragent	The website uses this package to get information about the requester.	`var useragent = require("useragent");`
			`var headers =`
			`  req.headers["user-agent"];`
			`var agent =`
			`  useragent.parse(headers);`

FIGURE 5.6: Usage scenarios we hypothesize the vulnerable modules to be involved in.

ID	Module	Vulnerable regular expression	Header
1	charset	`/(?:charset|encoding)\s*=\s*['"]? *([\w\-]+)/i`	Content-Type
2	content	`/^([^\/]+\/[^\s;]+)(?:(?:\s*;\s*boundary=(?:"([^"]+)"|([^;"]+)))|(?:\s*;\s*[^=]+=(?:(?:"(?:[^"]+)")|(?:[^;"]+))))*$/i`	Content-Type
3	fresh	`/ *, */`	If-None-Match
4	forwarded	`/ *, */`	X-Forwarded-For
5	mobile-detect	`new RegExp("Dell.*Streak|Dell.*Aero|Dell.*Venue|DELL.*Venue Pro|Dell Flash|Dell Smoke|Dell Mini 3iX|XCD28|XCD35|\\b001DL\\b|\\b101DL\\b|\\bGS01\\b")`	User-Agent
6	platform	`/^ +| +$/g`	User-Agent
7	ua-parser-js	`/ip[honead]+(?:.*os\s([\w]+)*\slike\smac|;\sopera)/`	User-Agent
8	useragent	`/((?:[A-z0-9]+|[A-z\-]+ ?)?(?: the)?(?:[Ss][Pp][Ii][Dd][Ee][Rr]|[Ss]crape|[A-Za-z0-9-]*(?:[^C][^Uu])[Bb]ot|[Cc][Rr][Aa][Ww][Ll])[A-z0-9]*)(?:(?:[ \/]| v)(\d+)(?:\.(\d+)(?:\.(\d+))?)?)?/`	User-Agent

FIGURE 5.7: Vulnerable modules with their corresponding exploitable regular expression and the HTTP header used for mounting the exploit.

to ours could create exploits that might work for a wide range of websites with relatively little effort. For an attack targeted at a specific website, we believe that more complex scenarios could be built, e.g., involving multiple HTTP requests and domain knowledge. For example, the `marked` package provides a parser for the markdown format. By crafting a specific markdown document, an attacker can block the main loop for hours. However, to deploy the exploit, complex interactions with the server are needed. That is, the attacker needs to figure out which part of the website may use a markdown parser and how to provide a document that will be processed by the parser. We believe that such a scenario is realistic, but it requires an in-depth analysis of each website. We leave for future work to test this hypothesis. In this work, our goal is to assess the effect of exploits that can be deployed at a large scale. Therefore, we only consider very sim-

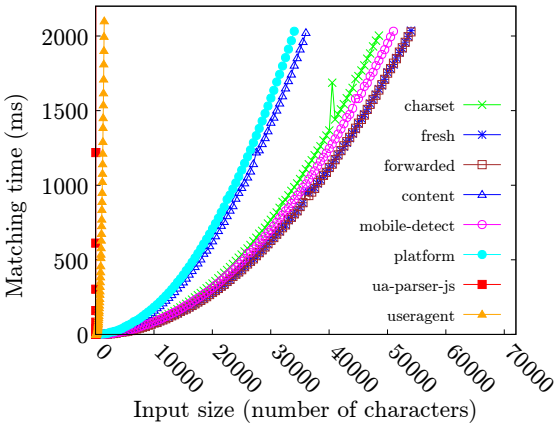


FIGURE 5.8: Matching time for different input sizes.

ple usage scenarios that can be triggered with a single HTTP request made to the main page.

To better understand the vulnerabilities, Figure 5.7 shows for each vulnerable module the vulnerable regular expressions. Some of the expressions are non-trivial, making it hard for developers to focus on possible ReDoS attacks in addition to the correctness of the regular expression. Four of these regular expressions can be successfully identified by a recent approach proposed by Wüstholz et al. [Wüs+17], which targets Java applications, though. The remaining four regular expressions cannot be detected by their approach due to differences between the regular expression semantics of Java and JavaScript.

5.4.2 *Matching Time*

We use the exploits to measure the influence of the size of the input to the matching time of the vulnerable expression (Figure 5.8). For most of the exploits, the input dependency seem to be quadratic, reaching one second matching time within 20,000 to 40,000 characters. For two exploits, the input dependency is presumably exponential, reaching one second matching time with less than 1,000 characters. We consider any of these eight exploits to be harmful because they may impact a website's availability (Section 5.4.3) and because even a non-exponential ReDoS vulnerability may aid an attacker in mounting a DoS attack (Section 5.5.1).

To further illustrate the effectiveness of inputs crafted for a specific regular expression, we measure the matching time for each vulnerable module with randomly created inputs. It turns out that random string inputs of the same size as our crafted exploits cause much lower matching times. The maximum matching time across the eight attacks is 20 milliseconds for inputs with 100,000 characters. We conclude that crafting inputs for vulnerable regular expressions is significantly more effective, from an attacker's perspective, than launching a brute-force DoS attack with randomly created inputs.

5.4.3 *Availability*

We now show that the matching time of a regular expression has a direct impact on the availability of a web server. To show the threat to availability posed by ReDoS exploits, we create a simple Express application with two features: it replies with a "hello world" message when called at the "/echo" path, and it calls the `forwarded` module with the request headers when called at the "/redos" path. We choose this module because it appears in Figure 5.8 to be the *least* harmful in our set of exploits, i.e., we are underestimating the negative impact on availability. We then upload this simple application on a machine running Node.js, provided by a commercial cloud platform[8].

We set up two other machines to concurrently send request. One machine, called the victim, measures the time it takes to trigger 100 requests of the "hello world" message. This victim machine triggers the next request once the previous request has been responded to. At the same time, the other machine, called the attacker, delivers 1,000 ReDoS payloads, by triggering all 1,000 requests at once. The victim machine starts its requests immediately after the victim machine has triggered its requests.

We vary the payload size from 0 characters to 8,000 characters in increments of 1,000 characters. A zero-sized payload is a request with an empty header instead of one that exploits the ReDoS vulnerability. We consider the zero-sized payload to check whether a Node.js server can be blocked using a brute-force strategy. We chose the upper limit for the payload size because, by default, the web server provider limits the size of the header fields to 8,500 characters. Other hosting providers allow significantly larger headers, as we report later in this section.

8 http://heroku.com

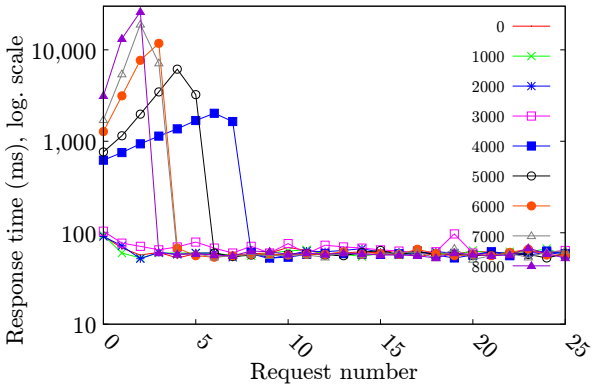


FIGURE 5.9: Impact of differently sized payloads on a server's response time. Note the logarithmic y-scale. Payloads are plotted in increments of 1,000 characters.

Figure 5.9 shows the response times measured at the victim machine for the first 25 "/echo" requests. Payloads smaller than 4,000 characters have no significant effect on the response time of the server. In contrast, payloads larger than this value delay as many as eight requests with a maximum delay of 20 seconds. By increasing the size of payloads, an attacker can control both the number of requests we delay and their duration. For the largest payloads we use, we even experienced dropping of requests.

This result is particularly remarkable because an individual payload of size 4,000 does not require an immense amount of time to respond to. We separately measured the CPU time required to respond to one such request and find it to take only 5.73 milliseconds, on average. However, several requests together can delay the victim's request by up to 20 seconds. This finding shows that the ReDoS payloads have a cumulative effect and even a small delay in the main loop can cause significant harm for availability.

We remind the reader that the above experiment uses the smallest payload in our data set, `forwarded`. Therefore, if we show that even this exploit poses a threat to availability, we can conclude that the rest of the exploits also do. For more severe vulnerabilities, e.g., in `ua-parser-js`, there is even no need to evaluate the impact on availability. As described in the Section 5.2, one single such payload is enough to completely block the server for as long as the matching takes. Considering that with 50–60 characters we predict a CPU computation time in the order of years, such vulnerabilities are a very serious threat to availability.

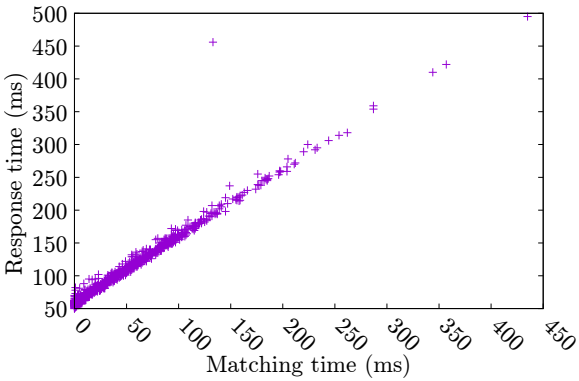


FIGURE 5.10: Correlation between server computation time and request response time.

5.4.4 *Response Time vs. Matching Time*

Our methodology relies on the assumption that small changes in the server computation time have an effect on clients. To validate this assumption we again use the `forwarded` package and the commercial web server setup from the previous section. We use 1,000 payloads smaller than 8,000 characters. The largest one of these payloads produces a matching time smaller than 100 milliseconds on our local machine. We measure the time spent by the server in the `forwarded` package and the time it takes for a request to be served at the client level. We then plot the relation between these two time measurements in Figure 5.10. The correlation between both measurements is 0.99, i.e., very strong. The strong correlation shows that the delays introduced by the network layer are relatively constant over time and that the server computation time is the dominant component in the response time measured at the client-side. Of course, the observed value depends on the chosen web server provider and the current server load, but we can safely conclude that measuring time at the client level is a good enough estimation of the server-side computation time.

5.4.5 *Dimensioning Exploits*

Choosing an appropriate size for the payload is a crucial part in our methodology and distinguishes our study from a real DoS attack on websites. The goal of this step is to find a payload size that is large enough

Module	P1: 100ms	P2: 200ms	P3: 500ms	P4: 1s	P5: 2s
fresh	12,000	17,000	27,000	37,500	53,500
forwarded	12,000	17,000	26,500	38,000	53,500
useragent	500	650	925	1,150	1,450
ua-parser-js	38	39	40	41	42
mobile-detect	10,500	15,500	25,000	36,500	50,500
platform	7,500	11,000	17,500	25,000	34,500
charset	10,500	15,500	24,000	34,000	48,000
content	8,000	11,000	18,000	25,500	35,500

FIGURE 5.11: Number of characters in each payload needed to achieve a specific delay in a vulnerable module.

to check whether a website is vulnerable to a specific attack, but small enough to only block the website for a negligible amount of time. To this end, we locally run each exploit five times with a payload of increasing size and stop the process when the matching time exceeds two seconds. We consider five target matching times, 100ms, 200ms, 500ms, 1s, and 2s, and choose the payload size that produces the closest matching time to the target time.

Figure 5.11 shows the values for each target time and vulnerable module. For example, for the `platform` vulnerability, we obtain a matching time of 200ms with a payload of 11,000 characters. The `useragent` and `ua-parser-js` packages, whose matching times grow at a much faster rate, requiring less than 1,500 characters to cause a delay of 2s.

5.4.6 *Vulnerable Sites*

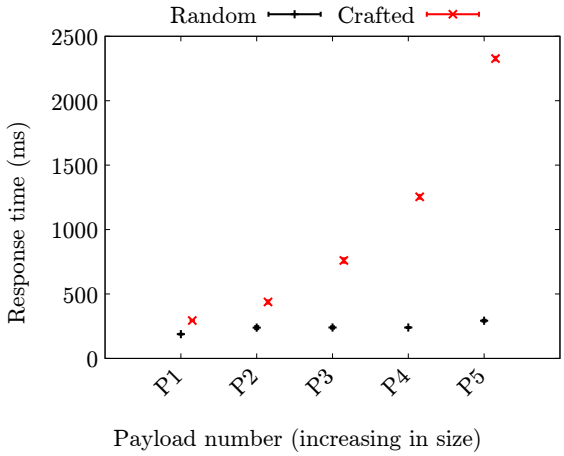
The goal of the next step is to assess to what extent real websites suffer from ReDoS vulnerabilities. Based on the five payload sizes for each exploit, we create attack payloads and random payloads for each exploit and payload size. We send these payloads to the 2,846 real websites that are running an Express webserver (Section 5.3.1). We warm up the connection three times and then measure five response times for both random and malicious inputs. Using the methodology described in Section 5.3.4, we then decide based on the measured response times whether a site is vulnerable. If for some reason, we could not send three or more out of the five payloads to a specific website, we consider that website to be non-vulnerable.

Overall, we observe that 339 sites suffer from at least one of the eight vulnerabilities. 66 sites actually suffer from two vulnerabilities and six sites even from three. This result shows that ReDoS attacks are a widespread problem that affects a large number of real-world websites. Given that our methodology is designed to underestimate the number of affected sites, e.g., because we consider only eight exploits, the actual number of ReDoS-vulnerable sites is likely to be even higher. Moreover, we expect the growing popularity of JavaScript on the server side to further increase the problem in the future.

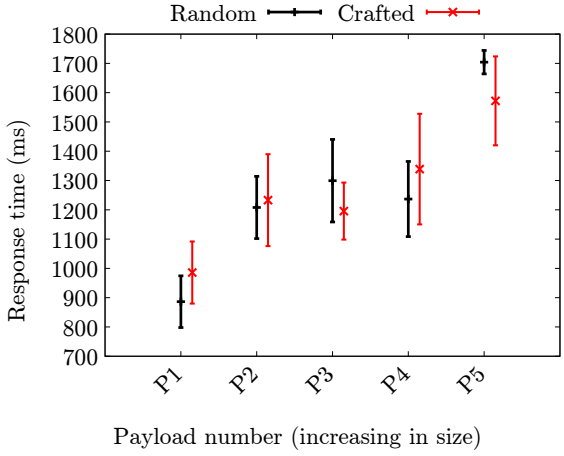
To illustrate our methodology for deciding whether a site is vulnerable, consider two example websites. In Figure 5.12, we plot for each of the five payload sizes the response time for malicious and random inputs. The figure shows the mean and the confidence intervals for a vulnerable site in Figure 5.12a and for a non-vulnerable site in Figure 5.12b. The response time grows significantly faster for the malicious payloads in the vulnerable site, reaching slightly more than two seconds for the fifth payload. In contrast, for the non-vulnerable site, the response time for both malicious and random payloads seems to grow linearly. Since the confidence interval for the response times in Figure 5.12b overlap, we classify this website as non-vulnerable. By inspecting other websites classified as vulnerable by our methodology, we observe patterns similar to Figure 5.12a. Therefore, we conclude that our criteria for deciding if a website is vulnerable are valid.

5.4.7 *Prevalence of Specific Vulnerabilities*

Figure 5.13 shows the number of websites affected by each vulnerability. Perhaps unsurprisingly, the vulnerabilities in `fresh` and `forwarded` have most impact, since these two modules are part of the Express framework. One of them needs to be activated using a configuration option, while the other module is enabled by default. One may ask why not all Express analyzed websites suffer from this problem. The reason is the way we dimension our payloads: Many Express instances limit the header size, and hence we cannot send large enough payloads to confirm that the sites are vulnerable. The other six vulnerabilities affect websites with a frequency that is roughly proportional to the popularity of the respective modules. For example, the vulnerability in the popular `useragent` affects more websites than the vulnerability in the less used `charset` module. To our initial surprise, we cannot confirm any site vulnerable due to the `content` mod-



(a) Response time for an vulnerable site.



(b) Response time for a non-vulnerable site.

FIGURE 5.12: Effect of increasing payload sizes on the response time of two websites.

Exploit	Affected sites
fresh	241
forwarded	99
ua-parser-js	41
useragent	16
mobile-detect	9
platform	8
charset	3
content	0

FIGURE 5.13: Number of websites affected by specific vulnerabilities.

ule. After more careful consideration, we realized that there are two more popular alternatives for parsing the `Content-Header` and the `content` package seems to be more popular among users of the `hapi.js` framework, which is a competitor of Express.

From an attacker's perspective, the distribution of vulnerabilities is great news, because exploits are portable across websites and knowing a vulnerabilities is sufficient to attack various websites. Likewise, the distribution is also good news for the community, showing that one can lower the risk of ReDoS in multiple websites by fixing a relatively small set of popular packages.

5.4.8 *Influence of Popularity*

Are ReDoS vulnerabilities a problem of less popular sites? In Figure 5.14, we show how the vulnerable sites are distributed across the Alexa top one million sites. For each point p on the horizontal axis, the vertical axis shows the number of exploitable sites with popularity rank $\leq p$. For example, there are 61 vulnerable sites in the top 100,000 websites, with one site in top 1,000 and nine in top 10,000. As can be observed from the distribution, the vulnerabilities are roughly equally distributed among the top one million sites. There is even a slight tendency toward more vulnerabilities among the more popular websites. This tendency can be explained by the trend we have seen in Figure 5.4, that server-side JavaScript tends to be more popular among popular websites. Overall, we can conclude that ReDoS vulnerabilities are a general problem that affects sites independent of their popularity ranking.

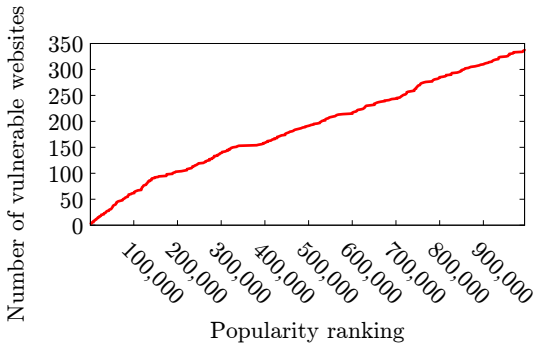


FIGURE 5.14: Cumulative distribution function showing the popularity of vulnerable sites. Each point on the graph shows how many sites among the top x sites suffer from at least one vulnerability.

5.4.9 *Use of Mitigation Techniques*

As mentioned before, some websites refuse to process a request whose header size exceeds a certain size. In Figure 5.15 we plot for each exploit how many websites accept a payload of a given size. As can be observed, most websites accept headers that are smaller than 10,000 characters, but only few websites accept headers that are, for instance, 40,000 characters long. As we have shown in Section 5.4.3, 10,000 characters are enough to do harm even with the least serious vulnerability. Therefore, the current limits that the websites apply on the header size are insufficient and they do not provide adequate protection against DoS.

Another interesting trend to observe in Figure 5.15 is that even for the most harmful exploit, `useragent`, for which we require payloads between 38 and 42 characters only, the number of websites that accept larger payloads decreases over time. This is surprising since for other exploits like `mobile-detect` there seem to be more websites to accept 10,000 characters long headers. We believe this observation to be due to the fact that some websites refuse to process many requests from the same user in a short period of time. For instance, our largest payload is sent after approximately 50 other requests of smaller size and the site refuses to serve it. This is a well known network-level protection against DoS, but there seem to be only around 200 websites to implement it. However, limiting the number of requests is no silver bullet against denial of service attacks, especially

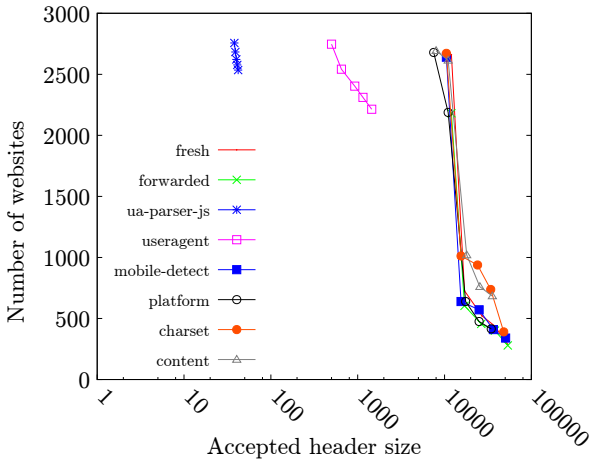


FIGURE 5.15: Number of websites that accept a payload of a specific size. Note the logarithmic x-scale.

when the attacker has the resources to deploy a distributed denial of service attack.

5.4.10 *Threats to Validity*

One threat to validity for our study is that we rely on time measurements performed over the network to estimate the likelihood of a ReDoS vulnerability. One may argue that these measurements should not be trusted and that pure chance made us observe some larger slowdowns for malicious payloads. We address this threat in multiple ways: We show that for commercial web hosting servers there is a high correlation between response time and server CPU time, we repeat measurements multiple times, and we draw conclusions only from statistically significant differences.

Another potential concern is that the exploits we created are too generic and happen to cause slowdown in another regular expression than the one we created them for. We believe that this situation would only impact our ability to tell which module is used on the server-side and not the impact of a ReDoS attack. Moreover, five of our exploits rely on a specific sequence of characters in the payload to the effective. These sequences of highly contextual characters need to be present in the beginning or at the end of the exploit. Removing any of them would make the exploit unusable.

Therefore, we believe that at least for these vulnerabilities it is very likely that our exploits indeed trigger the intended regular expression.

5.5 DISCUSSION

In this section, we discuss the potential of a large-scale DoS attack on NODE.JS websites and some defenses we recommend to minimize the impact of such an event. Finally, we describe an unexpected implication of our study: that algorithmic complexity attacks can be used for software fingerprinting.

5.5.1 *Impact of a Large-Scale Attack*

Compared to a regular DoS attack, a ReDoS vulnerability enables an attacker to launch an attack with fewer resources. As shown in Section 5.4.3, even the least harmful vulnerabilities we identify can be a lethal weapon when used as part of a large-scale DoS attack, because the attacker can send payloads that hang the loop for hundreds of milliseconds, several seconds, or even more, depending on the vulnerability. We remind the reader that with just eight standard attack vectors we could affect hundreds of websites.

It is worth emphasizing once again that this issue would not be as serious in a traditional thread-based web server, such as Apache. This is because the matching would be done in a thread serving the individual client. In contract, in an event-based system, the matching is done in the main loop and spending a few seconds matching a regular expression is equivalent to completely blocking the server for this amount of time.

A large-scale ReDoS attack against NODE.JS-based sites is a bleak scenario for which, as we have shown, many websites are not prepared. To limit this risk, we have been working with the maintainers of vulnerable modules to fix vulnerabilities. In addition, we urgently call for the adoption of multiple layers of defense, as outlined in the following.

5.5.2 *Defenses*

First of all, to limit the effect of a payload delivered through an HTTP header, the size of the header should be limited. For more than 15% sites, we could successfully deliver headers longer than 25,000 characters. We are not aware of any benign use cases for such large HTTP headers. Therefore,

a best practice in Node.js applications should be to limit the size of request headers. This kind of defense would mitigate the effects of some potential attacks, but is limited to vulnerabilities related to HTTP headers. In contrast, vulnerabilities related to other inputs received from the network, e.g., the body of an HTTP request, would remain exploitable.

Another defense mechanism could be to use a more sophisticated regular expression engine that guarantees linear matching time. The problem is that these engines do not support advanced regular expression features, such as look-ahead or back-references. Davis et al. [DKL17] advocate for a hybrid solution that only calls the backtracking engine when such advanced features are used, and to use a linear time algorithm in all other cases. This is an elegant solution that is already adopted by languages like Rust[9]. However, it would not completely solve the problem, since some regular expressions with advanced features may still contain ReDoS vulnerabilities. For instance, during our vulnerability study, we found the following regular expression:

```
/(?=.*\bAndroid\b)(?=.*\bMobile\b)/i
```

This expression from the `ismobilejs` module contains both lookahead and has super-linear complexity in a backtracking engine.

We also recommend that Node.js augments its regular expression APIs with an additional, optional timeout parameter. Node.js will stop any matching of regular expressions that takes longer than the specified timeout. This solution is far from perfect, but it is relatively easy to implement and adopt, has been successfully deployed in other programming languages [MTK12], and may also be feasible for Node.js [DWL18].

Additionally, we advocate that our work should be used as a roadmap for penetration testing sessions performed on Node.js websites. First, the tester audits the list of package dependencies, identifies any known ReDoS vulnerability in these packages or analyzes all the contained regular expressions. Second, the tester creates payloads for all the vulnerable regular expressions identified in the first step. Third, the tester tries to deliver these payloads using standard HTTP requests.

Finally, better tools and techniques should be created to help developers reason about ReDoS vulnerabilities in server-side JavaScript. Both static and dynamic analysis tools can aid in understanding the complexity of regular expressions and their performance. A good starting point could be

9 https://github.com/rust-lang/regex

porting existing solutions that were created for other languages, e.g., for Java [Wüs+17].

5.5.3 *Fingerprinting Web Servers*

Part of our methodology could be used for fingerprinting web servers, i.e., to predict some of the third-party modules used by a website. This ability can be useful for an attacker in at least two ways. First, the attacker may try to temper with the development process of that module by introducing backdoors that can then be exploited in the live website. Given that npm modules often depend on several others, the vulnerability can even be hidden in a dependent module. Second, the attacker may exploit a more serious vulnerability present in the same module. To show how this scenario may happen, consider the `dns-sync` vulnerability, identified in Section 5.4.1. The vulnerable function suffers both from a ReDoS and a command injection vulnerability, the later type discussed in detail in Chapter 4. An attacker may use the ReDoS attack as a hard-to-detect way to scan which sites use the vulnerable module and then attack these sites with a command injection.

5.6 CONCLUSIONS

This chapter studies ReDoS vulnerabilities in JavaScript-based web servers and shows that they are an important problem that affects various popular websites. We exploit eight vulnerabilities that affect at least 339 popular websites. We show that an attacker could block these vulnerable sites for several seconds and sometimes even much longer. More generally, our results are a call-to-arms to address the current lack of tools for analyzing ReDoS vulnerabilities in JavaScript.

6

LEAKY IMAGES ON THE CLIENT-SIDE

This chapter presents a novel client-side attack that exploits the way images are shared in popular websites. While the attack itself is not specific to full-stack JavaScript web applications, which are the object of study for this thesis, its automatic detection requires full-stack security analysis. We do not provide such a solution here, but we claim that it is much easier to build such a technique for full-stack JavaScript applications as opposed to more traditional, e.g., PHP-powered, ones. Therefore, the attack presented in this chapter serves as an example of sophisticated attacks that require full-stack analysis for automated detection. This chapter shares material with the corresponding publication [SP19].

6.1 MOTIVATION

Many popular websites allow users to privately share images with each other. For example, email services allow attachments to emails, most social networks support photo sharing, and instant messaging systems allow files to be sent as part of a conversation. We call websites that allow users to share images with each other *image sharing services*.

We present a targeted privacy attack that abuses a vulnerability we find to be common in popular image sharing services. The basic idea is simple yet effective: An attacker can determine whether a specific person is visiting an attacker-controlled website by checking whether the browser can access an image shared with this person. We call this attack *leaky images*, because a shared image leaks the private information about the victim's identity, which otherwise would not be available to the attacker. To launch a leaky images attack, the attacker privately shares an image with the victim through an image sharing service where both the attacker and the victim are registered as users. Then, the attacker includes a request for the image into the website for which the attacker wants to determine whether the victim is visiting it. Since only the victim, but no other user, is allowed to successfully request the image, the attacker knows with 100% certainty whether the victim has visited the site.

Threat	Who can attack?	What does the attacker achieve?	Usage scenario
Tracking pixels	Widely used ad providers and web tracking services	Learn that user visiting site A is the same as user visiting site B	Large-scale creation of low-entropy user profiles
Social media fingerprinting	Arbitrary website provider	Learn into which sites the victim is logged in	Large-scale creation of low-entropy user profiles
Cross-site request forgery	Arbitrary website provider	Perform side effects on a target site into which the victim is logged in	Abuse the victim's authorization by acting on her behalf
Leaky images	Arbitrary website provider	Precisely identify the victim	Targeted, fine-grained deanonymization

TABLE 6.1: Leaky images vs. related web attacks. All techniques assume that the victim visits an attacker-controlled website.

Beyond the basic idea of leaky images, we describe three further attacks. First, we describe a targeted attack against groups of users, which addresses the scalability issues of the single-victim attack. Second, we show a pseudonym linking attack that exploits leaky images shared via different image sharing services to determine which user accounts across these services belong to the same individual. Third, we present a scriptless version of the attack, which uses only HTML, and hence, works even for users who disable JavaScript in their browsers.

Leaky images can be (ab)used for targeted attacks in various privacy-sensitive scenarios. For example, law enforcement could use the attack to gather evidence that a suspect is visiting particular websites. Similarly but perhaps less noble, a governmental agency might use the attack to deanonymize a political dissident. As an example of an attack against a group, consider deanonymizing reviewers of a conference. In this scenario, the attacker would gather the email addresses of all committee members and then share leaky images with each reviewer through some of the various websites providing that service. Next, the attacker would embed a link to an external website into a paper under review, e.g., a link to a website with additional material. If and when a reviewer visits that page, while

being logged into one of the image sharing services, the leaky image will reveal to the attacker who is reviewing the paper. The prerequisite for all these attacks is that the victim has an account at a vulnerable image sharing service and that the attacker is allowed to share an image with the victim. We found at least three highly popular services (Google, Microsoft Live, and Dropbox) that allow sharing images with any registered user, making it straightforward to implement the above scenarios.

The leak is possible because images are exempted from the same-origin policy, and because image sharing services authenticate users through cookies. When the browser makes a third-party image request, it attaches the user's cookie of the image sharing website to it. If the decision of whether to authorize the image request is cookie-dependent, then the attacker can infer the user's identity by observing the success of the image request. Related work discusses the dangers of exempting JavaScript from the same-origin policy [Lek+15], but to the best of our knowledge, there is no work discussing the privacy implications of observing the result of cross-origin requests to privately shared images.

Leaky images differ from previously known threats by enabling arbitrary website providers to precisely identify a victim (Table 6.1). One related technique are tracking pixels, which enable tracking services to determine whether two visitors of different sites are the same user. Most third-party tracking is done by a few major players [EN16], allowing for regulating the way these trackers handle sensitive data. In contrast, our attack enables arbitrary attackers and small websites to perform targeted privacy attacks. Another related technique is social media fingerprinting, where the attacker learns whether a user is currently logged into a specific website.[1] In contrast, leaky images reveal not only whether a user is logged in, but precisely which user is logged in. Leaky images resemble cross-site request forgery (CSRF) [Shi], where a malicious website performs a request to a target site on behalf of the user. CSRF attacks typically cause side effects on the server, whereas our attack simply retrieves an image. We discuss in Section 6.5 under what conditions defenses proposed against CSRF, as well as other mitigation techniques, can reduce the risk of privacy leaks due to leaky images.

To understand how widespread the leaky images problem is, we study 30 out of the 250 most popular websites. We create multiple user accounts on these websites and check whether one user can share a leaky image with

1 See https://robinlinus.github.io/socialmedia-leak/ or https://browserleaks.com/social.

another user. The attack is possible if the shared image can be accessed through a link known to all users sharing the image, and if access to the image is granted only to certain users. We find that at least eight of the 30 studied sites are affected by the leaky images privacy leak, including some of the most popular sites, such as Facebook, Google, Twitter, and Dropbox. We carefully documented the steps for creating leaky images and reported them as privacy violations to the security teams of the vulnerable websites. In total, we informed eight websites about the problem, and so far, six of the reports have been confirmed, and for three of them we have been awarded bug bounties. Most of the affected websites are in the process of fixing the leaky images problem, and some of them, e.g., Facebook and Twitter, have already deployed a fix.

In summary, this chapter makes the following contributions:

- We present leaky images, a novel targeted privacy attack that abuses image sharing services to determine if a victim visits an attacker-controlled website.
- We discuss variants of the attack that aim at individual users, groups of users, that allow an attacker to link user identities across image sharing services, and that do not require any JavaScript.
- We show that eight popular websites, including Facebook, Twitter, Google, and Microsoft Live are affected by leaky images, exposing their users to be identified on third-party websites.
- We propose several ways to mitigate the problem and discuss their benefits and weaknesses.

6.2 IMAGE SHARING IN THE WEB

Many popular websites, including Dropbox, Google Drive, Twitter, and Facebook, enable users to upload images and to share these images with a well-defined set of other users of the same site. Let i be an image, U be the set of users of an image sharing service, and let $u^i_{owner} \in U$ be the owner of i. By default, i is not accessible to any other users than u^i_{owner}. However, an owner of an image can share the image with a selected subset of other users $U^i_{shared} \subseteq U$, which we define to include the owner itself. As a result, all users $u \in U^i_{shared}$, but no other users of the service and no other web users, have read access to i, i.e., can download the image via a browser.

SECRET URLS To control which users can access an image, there are several implementation strategies. One strategy is to create a *secret URL* for each shared image, and to provide this URL only to users allowed to download the image. In this scenario, there is a set of URLs L^i (L stands for "links") that point to a shared image i. Any user who knows a URL $l^i \in L^i$ can download i through it. To share an image i with multiple users, i.e., $|U^i_{shared}| > 1$, there are two variants of implementing secret URLs. On the one hand, each user u may obtain a personal secret URL l^i_u for the shared image, which is known only to u and not supposed to be shared with anyone. On the other hand, all users may share the same secret URL, i.e., $L^i = \{l^i_{shared}\}$. A variant of secret URLs are URLs that expire after a given amount of time or after a given number of uses. We call these URLs session URLs.

AUTHENTICATION Another strategy to control who accesses an image is to authenticate users. In this scenario, the image sharing service checks for each request to i whether the request comes from a user in U^i_{shared}. Authentication may be used in combination with secret URLs. In this case, a user u may access an image i only if she knows a secret URL l^i and if she is authenticated as $u \in U^i_{shared}$. The most common way to implement authentication in image sharing services are cookies. Once a user logs into the website of an image sharing service, the website stores a cookie in the user's browser. When the browser requests an image, the cookie is sent along with the request to the image sharing service, enabling the server-side of the website to identify the user.

IMAGE SHARING IN PRACTICE Different real-world image sharing services implement different strategies for controlling who may access which image. For example, Facebook mostly uses secret URLs, which initially created confusion among users due to the apparent lack of access control[2]. Gmail relies on a combination of secret URLs and authentication to access images attached to emails. Deciding how to implement image sharing is a tradeoff between several design goals, including security, usability, and performance. The main advantage of using secret URLs only is that third-party content delivery networks may deliver images, without any cross-domain access control checks. A drawback of secret URLs is that they should not be used over non-secret channels, such as HTTP, since these channels are unable to protect the secrecy of requested URLs. The main ad-

2 https://news.ycombinator.com/item?id=13204283

vantage of authentication is to not require links to be secret, enabling them to be sent over insecure channels. On the downside, authentication-based access control makes using third-party content delivery networks harder, because cookie-based authentication does not work across domains.

SAME-ORIGIN POLICY The same-origin policy regulates to what extent client-side scripts embedded in a website can access the document object model (DOM). As a default policy, any script loaded from one origin is not allowed to access parts of the DOM loaded from another origin. Origin here means the URI scheme (e.g., *http*), the host name (e.g., *facebook.com*), and the port number (e.g., 80). For example, the default policy implies that a website *evil.com* that embeds an `iframe` from *facebook.com* cannot access those parts of the DOM that have been loaded from *facebook.com*. There are some exceptions to the default policy described above. One of them, which is crucial for the leaky images attack, are images loaded from third parties. In contrast to other DOM elements, a script loaded from one origin can access images loaded from another origin, including whether the image has been loaded at all. For the above example, *evil.com* is allowed to check whether an image requested from *facebook.com* has been successfully downloaded.

6.3 PRIVACY ATTACKS VIA LEAKY IMAGES

This section presents a series of attacks that can be mounted using leaky images. At first, we describe the conditions under which the attack is possible (Section 6.3.1). Then, we present a basic attack that targets individual users (Section 6.3.2), a variant of the attack that targets groups of users (Section 6.3.3), and an attack that links identities of an individual registered at different websites (Section 6.3.4). Next, we show that the attack relies neither on JavaScript nor CSS, but can be performed by a purely HTML-based website (Section 6.3.5). Finally, we discuss how leaky images compare to previous privacy-related issues, such as web tracking (Section 6.3.6).

6.3.1 *Attack Surface*

Our attack model is that an attacker wants to determine whether a specific victim is visiting an attacker-controlled website. This information is important from a privacy point of view and usually not available to operators of a website. An operator of a website may be able to obtain some

information about clients visiting the website, e.g., the IP and the browser version of the client. However, this information is limited, e.g., due to multiple clients sharing the same IP or the same browser version, and often insufficient to identify a particular user with high confidence. Moreover, privacy-aware clients may further obfuscate their traces, e.g., by using the Tor browser, which hides the IP and other details about the client. Popular tracking services, such as Google Analytics, also obtain partial knowledge about which users are visiting which websites. However, the use of this information is legally regulated, available to only a few tracking services, and shared with website operators only in anonymized form. In contrast, the attack considered here enables an arbitrary operator of a website to determine whether a specific person is visiting the website.

Leaky image attacks are possible whenever all of the following four conditions hold. First, we assume that the attacker and the victim are both users of the same image sharing service. Since many image sharing services provide popular services beyond image sharing, such as email or a social network, their user bases often cover a significant portion of all web users. For example, Facebook announced that it has more than 2 billion registered users[3], while Google reported to have more than 1 billion active Gmail users each month[4]. Moreover, an attacker targeting a specific victim can simply register at an image sharing service where the victim is registered. Second, we assume that the attacker can share an image with the victim. For many image sharing services, this step involves nothing more than knowing the email address or user name of the victim, as we discuss in more detail in Section 6.4. Third, we assume that the victim visits the attacker-controlled website while the victim's browser is logged into the image sharing service. Given the popularity of some image sharing services and the convenience of being logged in at all times, we believe that many users fulfill this condition for at least one image sharing service. In particular, in Google Chrome and the Android operating system, users are encouraged immediately after installation to login with their Google account and to remain logged in at all times.

The fourth and final condition for leaky images concerns the way an image sharing service determines whether a request for an image is from a user supposed to view that image. Table 6.2 shows a two-dimensional matrix of possible implementation strategies, based on the description of secret URLs and authentication-based access control in Section 6.2. In one

3 https://techcrunch.com/2017/06/27/facebook-2-billion-users/
4 https://www.businessinsider.de/gmail-has-1-billion-monthly-active-users-2016-2

	URL of image		
Authentication (e.g., cookies)	Publicly known	Secret URL shared among users	Per-user secret URL
Yes	(1) Leaky image	(2) Leaky image	(3) Secure
No	(4) Irrelevant	(5) Secure	(6) Secure

TABLE 6.2: Conditions that enable leaky image attacks.

dimension, a website can either rely on authentication or not. In the other dimension, the site can make an image available through a publicly known URL, a secret URL shared among the users allowed to access the image, or a per-user secret URL. Out of the six cases created by these two dimensions, five are relevant in practice. The sixth case, sharing an image via a publicly known URL without any authentication, would make the image available to all web users, and therefore is out of the scope of this work. The leaky image attack works in two of the five possible cases in Table 6.2, cases 1 and 2. Specifically, leaky images are enabled by sites that protect shared images through authentication and that either do not use secret URLs at all or that use a single secret URL per shared image. Section 6.4 shows that these cases occur in practice, and that they affect some of today's most popular websites.

6.3.2 *Targeting a Single User*

After introducing the prerequisites for leaky images, we now describe several privacy attacks based on them. We start with a basic version of the attack, which targets a single victim and determines whether the victim is visiting an attacker-controlled website. To this end, the attacker uploads an image i to the image sharing service and therefore becomes the owner of the image, i.e., $u_{attacker} = u^i_{owner}$. Next, the attacker configures the image sharing service to share i with the victim user u_{victim}. As a result, the set of users allowed to access the image is $U^i_{shared} = \{u_{attacker}, u_{victim}\}$. Then, the attacker embeds a request for i into the website s for which the attacker wants to determine whether the victim is visiting the site. Because images are exempted from the same-origin policy (Section 6.2), the attacker-controlled parts of s can determine whether the image gets loaded

```
<script>
window.onload = function() {
 var img = document.getElementById("myPic");
 img.src = "https://imgsharing.com/leakyImg.png";
 img.onload = function() {
   httpReq("evil.com", "is the target");
 }
 img.onerror = function() {
   httpReq("evil.com", "not the target");
 }
}
</script>
<img id="myPic">
```

FIGURE 6.1: Tracking code included in the attacker's website.

successfully and report this information back to the attacker. Once the victim visits s, the image request will succeed and the attacker knows that the victim has visited s. If any other client visits s, though, the image request fails because s cannot authenticate the client as a user in U^i_{shared}. We assume that the attacker does not visit s, as this might mislead the attacker to believe that the victim is visiting s.

Because the authentication mechanism of the image sharing service ensures that only the attacker and the victim can access the image, a leaky image attack can determine with 100% accuracy whether the targeted victim has visited the site. At the same time, the victim may not notice that she was tracked, because the image can be loaded in the background.

For example, Figure 6.1 shows a simple piece of HTML code with embedded JavaScript. The code requests a leaky image, checks whether the image is successfully loaded, and sends this information back to the attacker-controlled web server via another HTTP request. We assume `httpReq` is a method that performs such a request using standard browser features such as `XMLHttpRequest` or `innerHTML` to send the value of the second argument to the domain passed as first argument. Alternatively to using `onload` to detect whether the image has been loaded, there are several variations, which, e.g., checking the width or height of the loaded image. As we show below (Section 6.3.5), the attack is also possible within a purely HTML-based website, i.e., without JavaScript.

The described attack works because the same-origin policy does not apply to images. That is, the attacker can include a leaky image through a cross-origin request into a website and observe whether the image is acces-

sible or not. In contrast, requesting an HTML document does not cause a similar privacy leak, since browsers implement a strict separation of HTML coming from different origins. A second culprit for the attack's success is that today's browsers automatically include the victim's cookie in third-party image requests. As a result, the request passes the authentication of the image sharing service, leaking the fact that the request comes from the victim's browser.

6.3.3 *Targeting a Group of Users*

The following describes a variant of the leaky images attack that targets a group of users instead of a single user. In this scenario, the attacker considers a group of n victims and wants to determine which of these victims is visiting a particular website.

For instance, consider a medium-scale spear phishing campaign against the employees of a company. After preparing the actual phishing payload, e.g., personalized emails or cloned websites, the attacker may include a set of leaky images to better understand which victims interact with the payload and in which way. In this scenario, leaky images provide a user experience analysis tool for the attacker.

A naive approach would be to share one image i_k ($1 \leq k \leq n$) with each of the n victims. However, this naive approach does not scale well to larger sets of users: To track a group of 10,000 users, the attacker needs 10,000 shared images and 10,000 image requests per visit of the website. In other words, this naive attack has $\mathcal{O}(n)$ complexity, both in the number of leaky images and in the number of requests. For the above example, this naive way of performing the attack might raise suspicion due to the degraded performance of the phishing site and the increase in the number of network requests.

To efficiently attack a group of users, an attacker can use the fact that image sharing services allow sharing a single image with multiple users. The basic idea is to encode each victim with a bit vector and to associate each bit with one shared image. By requesting the images associated with each bit, the website can compute the bit vector of a user and determine if the user is among the victims, and if yes, which victim it is. This approach enables a binary search on the group of users, as illustrated in Figure 6.2 for a group of seven users. The website includes code that requests images i_1, i_2, and i_3, and then determines based on the availability of the images which user among the targeted victims has visited the website. If none of

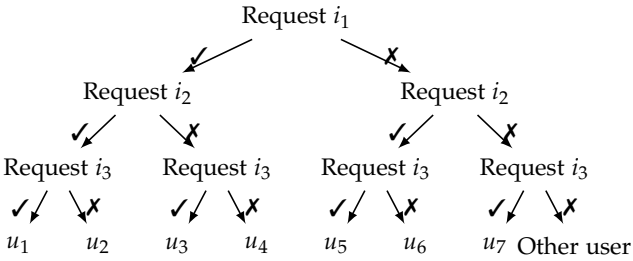


FIGURE 6.2: Binary search to identify individuals in a group of users u_1 to u_7 through requests to leaky images i_1 to i_3.

the images is available, then the user is not among the targeted victims. In contrast to the naive approach, the attack requires $\mathcal{O}(n)$ shared images and only $\mathcal{O}(log(n))$ image requests, enabling the attack on larger groups of users.

In practice, launching a leaky image attack against a group of users requires sharing a set of images with different subsets of the targeted users. This process can be automated, either through APIs provided by image sharing services or through UI-level web automation scripts. However, this process will most likely be website-specific which makes it expensive for attacking multiple websites at once.

6.3.4 *Linking User Identities*

The third attack based on leaky images aims at linking multiple identities that a single individual has at different image sharing services. Let *siteA* and *siteB* be two image sharing services, and let u_{siteA} and u_{siteB} be two user accounts, registered at the two image sharing services, respectively. The attacker wants to determine whether u_{siteA} and u_{siteB} belong to the same individual. For example, this attack might be performed by law enforcement entities to check whether a user account that is involved in criminal activities matches another user account that is known to belong to a suspect.

To link two user identities, the attacker essentially performs two leaky image attacks in parallel, one for each image sharing service. Specifically, the attacker shares an image i_{siteA} with u_{siteA} through one image sharing service and an image i_{siteB} with u_{siteB} through the other image sharing service. The attacker-controlled website requests both i_{siteA} and i_{siteB}. Once the

```
<!-- Three users (u1, u2, u3) have access to two
images (i1, i2) as follows: u1 to (i1);
u2 to (i2); u3 to (i1, i2) -->
<object data="leaky-domain.com/i1.png">
 <object data="evil.com?info=not_i1?sid=2342"/>
</object>
<object data="leaky-domain.com/i2.png">
 <object data="evil.com?info=not_i2?sid=2342"/>
</object>

<object data="leaky-domain.com/invalidImg.png">
 <object data="leaky-domain.com/invalidImg2.png">
  <object data="leaky-domain.com/invalidImg3.png">
   <object data="evil.com?info=loaded?sid=2342"/>
  </object>
 </object>
</object>
```

FIGURE 6.3: HTML-only variant of the leaky image group attack. All the `object` tags should have the `type` property set to `image/png`.

targeted individual visits this site, both requests will succeed and establish the fact that the users u_{siteA} and u_{siteB} correspond to the same individual. For any other visitors of the site, at least one request will fail because the two requests only succeed if the browser is logged into both user accounts u_{siteA} and u_{siteB}.

The basic idea of linking user accounts generalizes to more than two image sharing services and to user accounts of more than a single individual. For example, by performing two attacks on groups of users, as described in Section 6.3.3, in parallel, an attacker can establish pairwise relationships between the two groups of users.

6.3.5 *HTML-only Attack*

The leaky image attack is based on the ability of a client-side website to request an image and to report back to the attacker-controlled server-side whether the request was successful or not. One way to implement it is using client-side JavaScript code, as shown in Figure 6.1. However, privacy-aware users may disable JavaScript completely or use a security mechanism that prevents JavaScript code from reading details about images loaded from different domains.

We present a variant of the leaky image attack implemented using only HTML code, i.e., without any JavaScript or CSS. The idea is to use the `object` HTML tag, which allows a website to specify fallback content to be loaded if there is an error in loading some previously specified content.[5] When nesting such `object` elements, the browser first requests the resource specified in the outer element, and in case it fails, it performs a request to the inner element instead. Essentially, this behavior corresponds to a logical *if-not* instruction in pure HTML which an attacker may use to implement the leaky image attack.

Figure 6.3 shows an example of this attack variant. We assume that there are three users u_1, u_2, and u_3 in the target group and that the attacker can share leaky images from leaky-domain.com with each of them. The comment at the beginning of Figure 6.3 specifies the exact sharing configuration. We again need $log(n)$ images to track n users, as for the JavaScript-based attack against a group of users (Section 6.3.3). We assume that the server-side generates the attack code upon receiving the request, and that the generated code contains a session ID as part of the reporting links pointing to evil.com. In the example, the session ID is 2342. Its purpose is to enable the server-side code to link multiple requests coming from the same client.

The main insight of this attack variant is to place a request to the attacker's domain as fallbacks for leaky image requests. For example, if the request to the leaky image i_1 at line 4 fails, a request is made to evil.com for an alternative resource in line 5. This request leaks the information that the current user cannot access i_1, i.e., `info=not_i1`. By performing similar requests for all the leaky images, the attacker leaks enough information for precisely identifying individual users. For example, if in a given session, evil.com receives `not_i1`, but not `not_i1`, the attacker can conclude that the user is u_2. Because the server-side infers the user from the absence of requests, it is important to ensure that the current tracking session is successfully completed before drawing any conclusions. Specifically, we must ensure that the user or the browser did not stop the page load before all the nested `object` tags were evaluated. One way to ensure this property is to add a sufficiently high number of nested requests to non-existent images in lines 11 to 13 followed by a request that informs the attacker that the tracking is completed, in line 14. The server discards every session that does not contain this last message.

5 https://html.spec.whatwg.org/multipage/iframe-embed-object.html#the-object-element

As a proof of concept, we tested the example attack and several variants of it in the newest Firefox and Chrome browsers and find the HTML-only attack to work as expected.

6.3.6 *Discussion*

TRACKING PIXELS Leaky images are related to the widely used tracking pixels, also called web beacons [Cah+16; Eng+15; Yu+16], but both differ regarding who learns about a user's identity. A tracking pixel is a small image that a website s loads from a tracker website s_{track}. The image request contains the user's cookie for s_{track}, enabling the tracker to recognize users across different page visits. As a result, the tracking service can analyze which pages of s users visit and show this information in aggregated form to the provider of s. If the tracker also operates services where users register, it can learn which user visits which site. In contrast, leaky images enable the operator of a site s to learn that a target user is visiting s, without relying on a tracker to share this information, but by abusing an image sharing service. As for tracking pixels, an attacker can deploy leaky image attacks with images of 1x1 pixel size to reduce its impact on page loading time.

FINGERPRINTING Browser fingerprinting techniques [Aca+13; Aca+14; CLW17; Eck10; LRB16; MS12; Nik+13] use high-entropy properties of web browsers, such as the set of installed fonts or the size of the browser window, to heuristically recognize users. Like fingerprinting, leaky images aim at undermining the privacy of users. Unlike fingerprinting, the attacks presented here enable an attacker to determine specific user accounts, instead of recognizing that one visitor is likely to be the same as another visitor. Furthermore, leaky images can determine a visitor's identity with 100% certainty, whereas fingerprinting heuristically relies on the entropy of browser properties.

TARGETED ATTACKS VERSUS LARGE-SCALE TRACKING Leaky images are well suited for targeted attacks [Blo+14b; Har+14; Mar+14; SE13], but not for large-scale tracking of millions of users. One reason is that leaky images require the attacker to share an image with each victim, which is unlikely to scale beyond several hundreds users. Another reason is that the number of image requests that a website needs to perform increases linearly with the number of targeted users, as discussed in Section 6.3.3.

Hence, instead of aiming at large-scale tracking in the spirit of tracking pixels or fingerprinting, leaky images are better suited to target (sets of) individuals. However, this type of targeted attacks is reported to be increasingly popular, especially when dealing with high-value victims [SE13].

6.4 LEAKY IMAGES IN POPULAR WEBSITES

The attacks presented in the previous section make several assumptions. In particular, leaky images depend on how real-world image sharing services implement access control for shared images. To understand to what extent popular websites are affected by the privacy problem discussed in this chapter, we systematically study the prevalence of leaky images. The following presents our methodology (Section 6.4.1), our main findings (Section 6.4.2), and discusses our ongoing efforts toward disclosing the detected problems in a responsible way (Section 6.4.3).

6.4.1 *Methodology*

SELECTION OF WEBSITES To select popular image sharing services to study, we examined the top 500 most popular websites, according to the "Top Moz 500" list[6]. We focus on websites that enable users to share data with each other. We exclude sites that do not offer an English language interface and websites that do not offer the possibility to create user accounts. This selection yields a list of 30 websites, which we study in more detail. Table 6.3 shows the studied websites, along with their popularity rank. The list contains all of the six most popular websites, and nine of the ten most popular websites. Many of the analyzed sites are social media platforms, services for sharing some kind of data, and communication platforms.

IMAGE SHARING One condition for our attacks is that an attacker can share an image with a victim. We carefully analyze the 30 sites in Table 6.3 to check whether a site provides an image sharing service. To this end, we create multiple accounts on each site and attempt to share images between these accounts using different channels, e.g., chat windows or social media shares. Once an image is shared between two accounts, we check if the two accounts indeed have access to the image. If this requirement is met, we check that a third account cannot access the image.

6 https://moz.com/top500

ACCESS CONTROL MECHANISM For websites that act as image sharing services, we check whether the access control of a shared image is implemented in a way that causes leaky images, as presented in Table 6.2. Specifically, we check whether the access to a shared image is protected by authentication and whether both users access the image through a common link, i.e., a link known to the attacker. A site that fulfills also this condition exposes its users to leaky image attacks.

6.4.2 *Prevalence of Leaky Images in the Wild*

Among the 30 studied websites, we identify a total of eight websites that suffer from leaky images. As shown in Table 6.3 (column "Leaky images"), the affected sites include the three most popular sites, Facebook, Twitter, and Google, and represent over 25% of all sites that we study. The following discusses each of the vulnerable sites in detail and explains how an attacker can establish a leaky image with a target user. Table 6.4 summarizes the discussion in a concise way.

FACEBOOK Images hosted on Facebook are in general delivered by content delivery networks not hosted at the facebook.com domain, but, e.g., at fbcdn.net. Hence, the fact that facebook.com cookie is not sent along with requests to shared images disables the leaky image attacks. However, we identified an exception to this rule, where a leaky image can be placed at `https://m.facebook.com/photo/view_full_size/?fbid=xxx`. The `fbid` is a unique identifier that is associated with each picture on Facebook, and it is easy to retrieve this identifier from the address bar of an image page. The attacker must gather this identifier and concatenate it with the leaky image URL given above. By tweaking the picture's privacy settings, the attacker can control the subset of friends that are authorized to access the image, opening the door for individual and group attacks. A prerequisite of creating a leaky image on Facebook is that the victim is a "friend" of the attacker.

TWITTER Every image sent in a private chat on Twitter is a leaky image. The victim and the attacker can exchange messages on private chats, and hence send images, if one of them checked "Receive direct messages from anyone" in their settings or if one is a follower of the other. An image sent on a private chat can only be accessed by the two participants, based on their login state, i.e., these images are leaky images. The attacker can

Rank	Domain	Leaky images	Confirmed	Fix	Bug bounty
1	**facebook.com**	**yes**	**yes**	**yes**	**yes**
2	**twitter.com**	**yes**	**yes**	**yes**	**yes**
3	**google.com**	**yes**	**yes**	**planned**	**no**
4	youtube.com	no			
5	instagram.com	no			
6	linkedin.com	no			
8	pinterest.com	no			
9	wikipedia.org	no			
10	**wordpress.com**	**yes**	**no**	**no**	**no**
15	tumblr.com	no			
18	vimeo.com	no			
19	flickr.com	no			
25	vk.com	no			
26	reddit.com	no			
33	blogger.com	no			
35	**github.com**	**yes**	**no**	**no**	**no**
39	myspace.com	no			
54	stumbleupon.com	no			
65	**dropbox.com**	**yes**	**yes**	**planned**	**yes**
71	msn.com	no			
72	slideshare.net	no			
91	typepad.com	no			
126	**live.com**	**yes**	**yes**	**planned**	**no**
152	spotify.com	no			
160	goodreads.com	no			
161	scribd.com	no			
163	imgur.com	no			
166	photobucket.com	no			
170	deviantart.com	no			
217	**skype.com**	**yes**	**yes**	**planned**	**no**

TABLE 6.3: List of analyzed websites, whether they suffer from leaky images, and how the respective security teams have reacted to our notifications about the privacy leak.

Domain	Prerequisites	Image sharing channel	Authentication mechanism
facebook.com	Victim and attacker are "friends"	Image sharing	(5), (2)
twitter.com	Victim and attacker can exchange messages	Private message	(2)
google.com	*None*	Google Drive document	(3), (2)
		Private message	
wordpress.com	Victim is a viewer of the attacker's private blog	Posts on private blogs	(2)
github.com	Victim and attacker share a private repository	Private repository	(3), (2)
dropbox.com	*None*	Image sharing	(3), (6), (2)
live.com	*None*	Shared folder on OneDrive	(3), (2)
skype.com	Victim and attacker can exchange messages	Private message	(2)

TABLE 6.4: Leaky images in popular websites, the attack's preconditions, the image sharing channel and the implemented authentication mechanism as introduced in Table 6.2

easily retrieve the leaky image URL from the conversation and include it in another page. A limitation of the attack via Twitter is that we are currently not aware of a way of sharing an image with multiple users at once.

GOOGLE We identified two leaky image channels on Google's domains: one in the thumbnails of Google Drive documents and one in Google Hangouts conversations. To share documents with the victim, an attacker only needs the email address of the victim, while in order to send Hangouts messages, the victim needs to accept the chat invitation from the attacker. The thumbnail-based attack is more powerful since it allows to easily add and remove users to the group of users that have access to an image. Moreover, by unselecting the "Notify people" option when sharing, the victim users are not even aware of this operation. An advantage of the Hangouts channel, though, is that the victim has no way to revoke its rights to the leaky image, once the image has been received in a chat, as opposed to Drive, where the victim can remove a shared document from her cloud.

WORDPRESS To create a leaky image via Wordpress, the attacker needs to convince the victim to become a reader of his blog, or the other way around. Once this connection is established, every image posted on the shared private blog is a leaky image between the two users. Fulfilling this strong prerequisite may require non-trivial social engineering.

GITHUB Private repositories on GitHub enable leaky images. Once the victim and the attacker share a repository, every committed image can be accessed through a link in the web interface, e.g., `https://github.com/johndoe/my-awesome-project/raw/master/car.jpg`. Only users logged into GitHub who were granted access to the repository *my-awesome-project* can access the image. To control the number of users that have access to the image, the attacker can remove or add contributors to the project.

DROPBOX Every image uploaded on Dropbox can be accessed through a leaky image endpoint by appending the HTTP parameter `dl=1` to a shared image URL. Dropbox allows the attacker to share such images with arbitrary email addresses and to fine-tune the permissions to access the image by including and excluding users at any time. Once the image is shared, our attack can be successfully deployed, without requiring the victim to accept the shared image. However, the victim can revoke its rights to access an image by removing it from the "Sharing" section of her account.

LIVE.COM Setting up a leaky image on One Drive, a cloud storage platform available on a live.com subdomain, is very similar to the other two file sharing services that we study, Google Drive and Dropbox. The attacker can share images with arbitrary email addresses and the victim does not need to acknowledge the sharing. Moreover, the attacker can easily deploy a group attack due to the ease in changing the group of users that have access to a particular image.

SKYPE In the Skype web interface, every image sent in a chat is a leaky image. Note that most of the users probably access the service through a desktop or mobile standalone client, hence the impact of this attack is limited to the web users. Moreover, Skype automatically logs out the user from time to time, limiting the time window for the attack.

Our study of leaky images in real-world sites enables several observations.

LEAKY IMAGES ARE PREVALENT The first and perhaps most important observation is that many of the most popular websites allow an attacker to create leaky images. From an attacker's point of view, a single leaky image is sufficient to track a user. If a victim is registered as a user with at least one of the affected image sharing services, then the attacker can create a user account at that service and share a leaky image with the victim.

VICTIMS MAY NOT NOTICE SHARING A LEAKY IMAGE Several of the affected image sharing services enable an attacker to share an image with a specific user without any notice given to the user. For example, if the attacker posts an image on her Facebook profile and tweaks the privacy settings so that only the victim can access it, then the victim is not informed in any way. Another example is Google Drive, which allows sharing files with arbitrary email addresses while instructing the website to not send an email that informs the other user.

VICTIMS CANNOT "UNSHARE" A LEAKY IMAGE For some services, the victim gets informed in some way that a connection to the attacker has been established. For example, to set up a leaky image on Twitter, the attacker needs to send a private message to the victim, which may make the victim suspicious. However, even if the victim knows about the shared image, for most websites, there is no way for a user to revoke its right to access the image. Specifically, let's assume the victim receives a cute cat picture from a Google Hangouts contact. Let us now assume that the victim is aware of the leaky image attack and that she suspects the sender of the image tracking her. We are not aware of any way in which the victim can revoke the right to access the received image.

IMAGE SHARING SERVICES USE A DIVERSE MIX OF IMPLEMENTATION STRATEGIES Secret URLs and per-user authenticated URLs are widely implemented techniques that protects against our attack. However, many websites use multiple such strategies and hence, it is enough if one of the API endpoints uses leaky images. Identifying this endpoint is often a hard task: for example, in the case of Facebook, most of the website rigorously implements secret URLs, but one API endpoint belonging to a mobile subdomain exposes leaky images. After identifying this endpoint we realized that it can be accessed without any problem from a desktop browser as well, enabling all the attacks we describe in Section 6.3.

THE ATTACK SURFACE VARIES FROM SITE TO SITE Some but not all image sharing services require a trust relation between the attacker and the victim before a leaky image can be shared. For example, an attacker must first befriend a victim on Facebook before sharing an image with the victim, whereas no such requirement exists on Dropbox or Google Drive. However, considering that most users have hundreds of friends on social networks, there is a good chance that a trust channel is established before the attack starts. In the case of Wordpress the prerequisite that the "victim is a viewer of the attacker's private blog" appears harder to meet and may require advanced social engineering. Nonetheless, we believe that such leaky images may still be relevant in certain targeted attacks. Moreover, three of the eight vulnerable sites allow attackers to share images with arbitrary users, without any prerequisites (Table 6.4).

Since our study of the prevalence of leaky images is mostly manual, we cannot guarantee that the 22 sites for which we could not create a leaky image are not affected by the problem. For some sites, though, we are confident that they are not affected, as these sites do not allow users to upload images. A more detailed analysis would require in-depth knowledge of the implementation of the studied sites, and ideally also access to the server-side source code. We hope that our results will spur future work on more automated analyses that identify leaky images.

6.4.3 *Responsible Disclosure and Feedback from Websites*

After identifying image sharing services that suffer from leaky images, we contacted their security teams to disclose the problem in a responsible way. Between March 26 and March 29, 2018, we sent a detailed description of the general problem, how the specific website can be abused to create leaky images, and how it may affect the privacy of users of the site. Most security teams we contacted were very responsive and eager to collaborate upon fixing the issue.

CONFIRMED REPORTS The last three columns of Table 6.3 summarize how the security teams of the contacted companies reacted to our reports. For most of the websites, the security teams confirmed that the reported vulnerability is worth fixing, and at least six of the sites have already fixed the problem or have decided to fix it. In particular, the top three websites all confirmed the reported issue and all have been working on fixing it. Given the huge user bases of these sites and the privacy implications of leaky

images for their users, this reaction is perhaps unsurprising. As another sign of appreciation of our reports, the authors have received bug bounties from (so far) three of the eight affected sites.

DISMISSED REPORTS Two of our reports were flagged as false positives. The security teams of the corresponding websites replied by saying that leaky images are a "desired behavior" or that the impact on privacy of their user is limited. Comparing Table 6.3 with Table 6.4 shows that the sites that dismiss our report are those where the prerequisites for creating a leaky image are harder to fulfill than for the other sites: Creating a leaky image on GitHub requires the attacker and the victim to share a private repository, and Wordpress requires that the victim is a viewer of the attacker's private blog. While we agree that the attack surface is relatively small for these two sites, leaky images may nevertheless cause surprising privacy leaks. For example, an employee might track her colleagues or even her boss if their company uses private GitHub repositories.

CASE STUDY: FIX BY FACEBOOK To illustrate how image sharing services may fix a leaky images problem, we describe how Facebook addressed the problem in reaction to our report. As mentioned earlier, Facebook employs mostly secret URLs and uses content delivery networks to serve images. However, we were able to identify a mobile API endpoint that uses leaky images and redirects the user to the corresponding content delivery network link. This endpoint is used in the mobile user interface for enabling users to download the full resolution version of an image. The redirection was performed at HTTP level, hence it resulted in a successful image request when inserted in a third-party website using the `<a>` HTML tag. The fix deployed by Facebook was to perform a redirection at JavaScript level, i.e. load an intermediate HTML that contains a JavaScript snippet that rewrites `document.location.href`. This fix enables a benign user to still successfully download the full resolution image through a browser request, but disables third-party image inclusions. However, we believe that such a fix does not generalize and cannot be deployed to the other identified vulnerabilities. Hence, we describe alternative ways to protect against a leaky image attacks in Section 6.5.

CASE STUDY: FIX BY TWITTER A second case study of how websites can move away from leaky images comes from Twitter that changed its API[7]

7 https://twitter.com/TwitterAPI/status/1039631353141112832

in response to our report[8]. First, they disabled cookie-based authentication for images. Second, they changed the API in a way that image URLs are only delivered on secure channels, i.e., only authenticated HTTPS requests. Last, Twitter also changed the user interface to only render images from strangers when explicit consent is given. Essentially, Twitter moved from implementation strategy (2) to (5) in Table 6.2 in response to our report.

Overall, we conclude from our experience of disclosing leaky images that popular websites consider it to be a serious privacy problem, and that they are interested in detecting and avoiding leaky images.

6.5 MITIGATION TECHNIQUES

In this section, we describe several techniques to defend against leaky image attacks. The mitigations range from server-side fixes that websites can deploy, over improved privacy settings that empower users to control what is shared with them, to browser-based mitigations.

6.5.1 *Server-Side Mitigations*

The perhaps easiest way to defend against the attack presented in this chapter is to modify the server-side implementation of an image sharing service, so that it is not possible anymore to create leaky images. There are multiple courses of actions to approach this issue.

First, a controversial fix to the problem is to disable authenticated image requests altogether. Instead of relying on, e.g., cookies to control who can access an image, an image sharing service could deliver secret links only to those users that should access an image. Once a user knows the link she can freely get the image through the link, independent of whether she is logged into the image sharing service or not. This strategy corresponds to case 5 in Table 6.2. Multiple websites we report about in Table 6.3 implement such an image sharing strategy. The most notable examples are Facebook, which employs this technique in most parts of their website, and Dropbox, which implements this technique as part of their link sharing functionality. The drawback of this fix is that the link's secrecy might be compromised in several ways outside of the control of the image sharing service: by using insecure channels, such as HTTP, through side-channel attacks in the browser, such as cache attacks [Koc+19], or simply by having

8 https://hackerone.com/reports/329957

the users handle the links in an insecure way because they are not aware of the secrecy requirements.

Second, an alternative fix is to enforce an even stricter cookie-based access control on the server-side. In this case, the image sharing service enforces that each user accesses a shared image through a secret, user-specific link that is not shared between users. As a result, the attacker does not know which link the victim could use to access a shared image, and therefore the attacker cannot embed such a link in any website. This implementation strategy corresponds to case 3 in Table 6.2. On the downside, implementing this defense may prove challenging due to the additional requirement of guaranteeing the mapping between users and URLs, especially when content delivery networks are involved. Additionally, it may cause a slowdown for each image request due to the added access control mechanism.

Third, one may deploy mitigations against CSRF.[9] One of them is to use the `origin` HTTP header to ensure that the given image can only be embedded on specific websites. The `origin` HTTP header is sent automatically by the browser with every request, and it precisely identifies the page that requests a resource. The server-side can check the request's `origin` and refuse to respond with an authenticated image to unknown third-party request. For example, facebook.com could refuse to respond with a valid image to an HTTP request with the `origin` set to evil.com. However, this mitigation cannot defend against tracking code injected into a trusted domain. For example, until recently Facebook allowed users to post custom HTML code on their profile page. If a user decides to insert leaky image-based tracking code on the profile page, to be notified when a target user visits the profile page, then the CSRF-style mitigation does not prevent the attack. The reason for this is that the request's origin would be set to facebook.com, and hence the server-side code will trust the page and serve the image.

Similarly, the server can set the `Cross-Origin-Resource-Policy` response header on authenticated image requests and thus limit which websites can include a specific image. Browsers will only render images for which the origin of the request matches the origin of the embedding website or if they correspond to the same site. This solution is more coarse-grained than the previously discussed `origin` checking since it does not allow for cross-origin integration of authenticated images, but it is easier

9 https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

to deploy since it only requires a header set instead of a header check. The `From-Origin` header was proposed for allowing a more fine-grained integration policy, but to this date there is no interest from browser vendors side to implement such a feature.

Another applicable CSRF mitigation is the `SameSite` cookie attribute. When set to "strict" for a cookie, the attribute prevents the browser from sending the cookie along with cross-site requests, which effectively prevents leaky images. However, the "strict" setting may be too strict for most image sharing services, because it affects all links to the service's website. For example, a link in a corporate email to a private GitHub project or to a private Google Doc would not work anymore, because when clicking the link, the session cookie is not sent along with the request. The less restrictive "lax" setting of the `SameSite` attribute does not suffer from these problems, but it also does not prevent leaky images attacks, as it does not affect GET requests.

A challenge with all the described server-side defenses is that they require the developers to be aware of the vulnerability in the first place. From our experience, a complex website may allow sharing images in several ways, possibly spanning different UI-level user interactions and different API endpoints supported by the image sharing service. Since rigorously inspecting all possible ways to share an image is non-trivial, we see a need for future work on automatically identifying leaky images. At least parts of the methodology we propose could be automated with limited effort. To check whether an image request requires authentication, one can perform the request in one browser where the user is logged in, and then try the same request in another instance of the browser in "private" or "incognito" mode, i.e., without being logged in. Comparing the success of the two requests reveals whether the image request relies in any form of authentication, such as cookies. Automating the rest of our methodology requires some support by image sharing services. In particular, automatically checking that a leaky image is accessible only by a subset of a website's users, requires APIs to handle user accounts and to share images between users.

Despite the challenges in automatically identifying leaky images, we believe that server-side mitigations are the most straightforward solution, at least in the short term. In the long term, a more complete solution would be desirable, such as those described in the following.

6.5.2 *Browser Mitigations*

The current HTTP standard does not specify a policy for third-party cookies[10], but it encourages browser vendors to experiment with different such policies. More precisely, the standard lets the browser decide whether to automatically attach the user's cookie to third-party requests. Most browsers decide to attach third-party cookies, but there are certain counter-examples, such as the Tor browser. In Tor, cookies are sent only to the domain typed by the user in the address bar.

Considering the possible privacy implications of leaky images and other previously reported tracking techniques [Cah+16], one possible mitigation would be that browsers specify as default behavior to not send cookies with third-party (image) requests. If this behavior is overwritten, possibly using a special HTTP header or tag, the user should be informed through a transparent mechanism. Moreover, the user should be offered the possibility to prevent the website from overwriting the default behavior. We believe this measure would be in the spirit of the newly adopted European Union's General Data Protection Regulation which requires *data protection by design and by default*. However, such an extreme move may impact certain players in the web ecosystem, such as the advertisement ecosystem. To address this issue, advertisers may decide to move towards safer distribution mechanisms, such as the one popularized by the Brave browser.

An alternative to the previously discussed policy is to allow authenticated image requests, but only render them if the browser is confident that there are no observable differences between an authenticated request and a non-authenticated one. To this end, the browser could perform two image requests instead of one: one request with third-party cookies and one request without. If the browser receives two equivalent responses, it can safely render the content, since no sensitive information is leaked about the authenticated user. This solution would still allow most of the usages of third-party cookies, e.g. tracking pixels, but prevent the leaky image attack described here. A possible downside might be the false positives due to strategy (3) in Table 6.2, but we hypothesize that requests to such images rarely appear in benign third-party image requests. A second possible drawback of this solution may be the increase in web traffic and the potential performance penalties. Future work should test the benefits of this defense and the cost imposed by the additional image request.

10 https://tools.ietf.org/html/rfc6265#page-28

To reduce the cost imposed by an additional image request, a hybrid mechanism could disable authenticated image requests by default, and allow them only for the resources specified by a CSP directive. For the allowed authenticated images, the browser deploys the double image requests mechanism described earlier. We advocate this as our preferred browser-level defense since it can also defend against other privacy attacks, e.g. reading third-party image pixels through a side channel [Kot+13], while still permitting benign uses.

Similarly to ShareMeNot [RKW12], one can also implement a browser mechanism in which all third-party image requests are blocked unless the user authorizes them by providing explicit consent. To release the burden from the user, a hybrid mechanism can be deployed in which the website requires authenticated requests only for a subset of images for which the user needs to provide consent.

Another solution for when third-party cookies are allowed is for the browsers to implement some form of information flow control to ensure that the fact whether a third-party request was successfully loaded or not, cannot be sent outside of the browser. A similar approach is deployed in *tainted canvas*[11], which disallows pixel reads after a third-party image is painted on the canvas. Implementing such an information flow control for third-party images may, however, be challenging in practice, since the fact whether an image has successfully loaded or not can be retrieved through multiple side channels, such as the `object` tag or by reading the size of the contained div.

The mechanisms described in this section vary both in terms of implementation effort required for deploying them and in terms of their possible impact on the existing state of the web, i.e., incompatibility with existing websites. Therefore, to aid the browser vendors to take an informed decision, future work should perform an in-depth analysis of all these defenses in terms of usability, compatibility and deployment cost, in the style of Calzavara et al. [Cal+17], and possibly propose additional solutions.

6.5.3 *Better Privacy Control for Users*

A worrisome finding of our prevalence study is that a user has little control over the image sharing process. For example, for some image sharing services, the user does not have any option to restrict which other users

11 https://html.spec.whatwg.org/multipage/canvas.html#security-with-canvas-elements

can privately share an image with her. In others, there is no way for a user to revoke her right to access a specific image. Moreover, in most of the websites we analyzed, it is difficult to even obtain a complete list of images privately shared with the current account. For example, a motivated user who wants to obtain this list must check all the conversations in a messaging platform, or all the images of all friends on a social network.

We believe that image sharing services should provide users more control over image sharing policies, to enable privacy-aware users to protect their privacy. Specifically, a user should be allowed to decide who has the right to share an image with her and she should be granted the right to revoke her access to a given image. Ideally, websites would also offer the user a list of all the images shared with her and a transparent notification mechanism that announces the user when certain changes are made to this list. Empowering the users with these tools may help mitigate some of the leaky image attacks by attracting user's attention to suspicious image sharing, allowing users to revoke access to leaky images.

The privacy controls for web users presented in this section will be useful mostly for advanced users, while the majority of the users are unlikely to take advantage of such fine-grained controls. Therefore, we believe that the most effective mitigations against leaky images are at the server side or browser level.

6.6 CONCLUSIONS

This chapter presents leaky images, a targeted deanonymization attack that leverages specific access control practices employed in popular websites. The main insight of the attack is a simple yet effective observation: Privately shared resources that are exempted from the same origin policy can be exploited to reveal whether a specific user is visiting an attacker-controlled website. We describe several flavors of this attack: targeted tracking of single users, group tracking, pseudonym linking, and an HTML-only attack.

We show that some of the most popular websites suffer from leaky images, and that the problem often affects any registered users of these websites. We reported all the identified vulnerabilities to the security teams of the affected websites. Most of them acknowledge the problem and some already proceeded to fixing it. This feedback shows that the problem we identified is important to practitioners. Our work helps raising awareness among developers and researchers to avoid this privacy issue in the future.

Part III

Defenses

7

DEFENDING AGAINST INJECTION ATTACKS

Chapter 4 shows that the risk of injection vulnerabilities is widespread, and that a practical technique to mitigate them must support module maintainers who are not particularly responsive. Given these conclusions about the development process around Node.js modules, it is crucial to offer a solution that provides complete automation. This chapter presents Synode, an automatic approach to identify potential injection vulnerabilities and to prevent injection attacks. To the best of our knowledge, our approach is the first to address the problem of injection vulnerabilities in Node.js JavaScript code. This chapter shares material with the corresponding publication [SPL18].

The basic idea of Synode is to check all third-party modules as part of their *installation* and to rewrite them to enable a *safe mode* proposed in this chapter. A mix of two strategies is applied as part of rewriting.

- **Static:** we *statically* analyze the values that may be passed to APIs prone to injections. The static analysis also extracts a *template* that describes values passed to these dangerous APIs.
- **Runtime:** for code locations where the static analysis cannot ensure the absence of injections, we offer a *dynamic* enforcement mechanism that stops malicious inputs before passing them to the APIs.

A combination of these techniques is applied to a module at the time of installation via Node.js installation hooks, effectively enforcing a safe mode for third-party modules. In principle, while our runtime enforcement may be overly conservative, our evaluation in Section 7.5 shows that such cases are rare.

Alternative approaches: There are several alternatives to our hybrid analysis. One alternative is a sound static analysis that conservatively rejects all Node.js modules for which the analysis cannot guarantee the absence of injection vulnerabilities. Unfortunately, due to the dynamic features of JavaScript [And+17], a reasonably precise static analysis is virtually never sound [And+17; Wei15], whereas a fully sound analysis would result in many false positives. Another alternative is a training-based approach that learns from safe executions which values are safe to pass

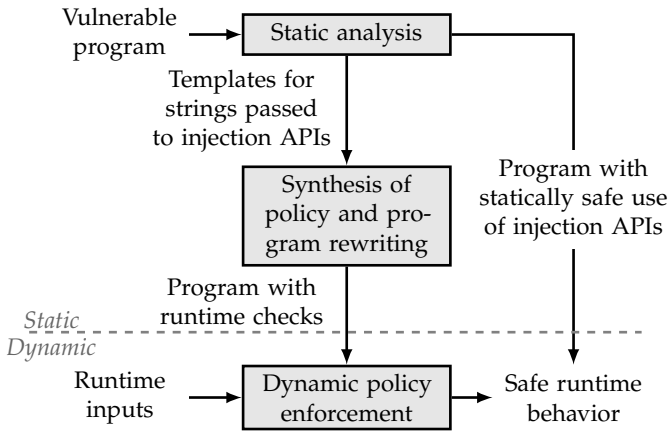


FIGURE 7.1: Architectural diagram of SYNODE.

to APIs prone to injections. While this approach works well for client-side JavaScript [Mit+16; Pan+16; SBS15], where a lot of runtime behavior gets triggered by loading the page, relying on training is challenging for NODE.JS code, which often comes without any inputs to execute the code. Finally, security-aware developers could resort to manually analyzing third-party modules for potential vulnerabilities. However, manual inspection does not scale well to the large number of modules available for NODE.JS and suffers from human mistakes. Instead of these alternatives, SYNODE takes a best-effort approach in the spirit of soundiness [Liv+15] that guarantees neither to detect all vulnerabilities nor to reject only malicious inputs. Our experimental evaluation shows that, in practice, SYNODE rejects very few benign inputs and detects all malicious inputs used during the evaluation.

SYNODE relates to existing work aimed at discovering and preventing code injections in JavaScript [GL09; LV09; SML11]. The closest existing approaches, Blueprint [LV09] and ScriptGard [SML11], share the idea of restricting runtime behavior based on automatically inferred templates. In contrast to them, SYNODE infers templates statically, i.e., without relying on inputs that drive the execution during template inference. Our work differs from purely static approaches [GL09] by defering some checks to runtime instead of rejecting potentially benign code. Moreover, all existing work addresses XSS vulnerabilities, whereas we address injection attacks on NODE.JS code.

Contributions:

- **Static analysis**: We present a static analysis that attempts to infer templates of user inputs used at potentially dangerous sinks. (Section 7.2)
- **Runtime enforcement**: For cases that cannot be shown safe via static analysis, we present a runtime enforcement achieved through code rewriting. The runtime approach uses partially instantiated abstract syntax trees (ASTs) and ensures that the runtime values do not introduce any unwanted code beyond what is expected. (Section 7.3.2)
- **Evaluation**: We apply our static technique to a set of 15,604 Node.js modules that contain calls to sinks. We discover that 36.66% of the sink call sites are statically guaranteed to be safe. For a subset of the statically unsafe modules, we create both malicious inputs that exploit the injection vulnerabilities and benign inputs that exercise the advertised functionality of the module. Our runtime mechanism effectively prevents 100% of the attacks, while being overly conservative for only 8.92% of the benign inputs.

Our implementation and a benchmark suite containing both malicious and benign inputs passed to the vulnerable modules is available for download:

https://github.com/sola-da/Synode

7.1 Methodology

The overall idea of the mitigation technique is to prevent injections at the call sites of injection APIs. Figure 7.1 shows an overview of the approach. Given a potentially vulnerable JavaScript module, a static analysis summarizes the values that may flow into injection APIs in the form of string templates, or short *templates*. A template is a sequence consisting of string constants and holes to be filled with untrusted runtime data. For call sites where the analysis can statically show that no untrusted data may flow into the injection API, no further action is required to ensure safe runtime behavior. Similar approaches for identifying statically safe call sites are adopted in practice for other languages, e.g., Java[1] and Python[2].

1 https://www.youtube.com/watch?v=ccfEu-Jj0as
2 https://github.com/dropbox/python-invariant

For the remaining call sites, the approach synthesizes a runtime check and statically rewrites the source code to perform this check before untrusted data reaches the injection API. When executing the module, the rewritten code enforces a security policy that checks the runtime values to be filled into the holes of the template against the statically extracted template. If this check fails, the program is terminated to prevent an injection attack.

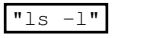
The Synode approach is conservative in the sense that it prevents potential vulnerabilities without certain knowledge of whether a vulnerability gets exploited by an attacker. The reason for this design decision is twofold. First, most Node.js modules are used in combination with other modules, i.e., we cannot reason about the entire program. Second, there is no trust model that specifies which module should sanitize untrusted inputs or even which inputs are untrusted. Our assumption is that user inputs and inputs from other modules are potentially attacker-controlled, an assumption shared by the vulnerabilities published at the Node Security Platform. Given these constraints and assumptions, Synode protects users of potentially vulnerable modules in an automatic way.

7.2 Static Analysis

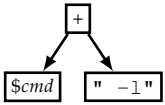
We present a static analysis of values passed to injection APIs. For each call site of such an API, the analysis summarizes all values that may be passed to the called function into a tree representation (Section 7.2.1). Then, the analysis statically evaluates the tree to obtain a set of templates, which represent the statically known and unknown parts of the possible string values passed to the function (Section 7.2.2). Finally, based on the templates, the analysis decides for each call site of an injection API whether it is statically safe or whether to insert a runtime check that prevents malicious strings from reaching the API (Section 7.2.3).

7.2.1 *Extracting Template Trees*

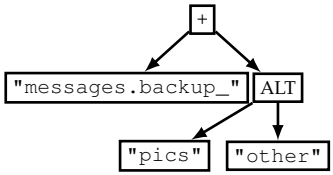
The analysis is a flow-sensitive, path-insensitive, intra-procedural, backward data flow analysis. Starting from a call site of an injection API, the analysis propagates information about the possible values of the string argument passed to the API call along inverse control flow edges. The propagated information is a tree that represents the current knowledge of the analysis about the value:



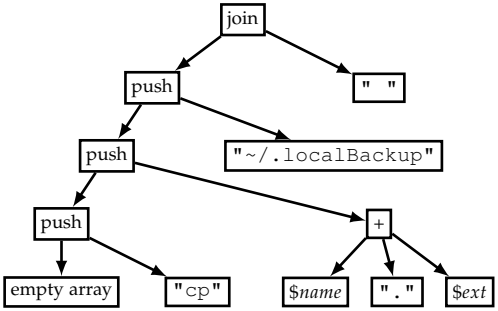
(a) Constant string.



(b) Concatenation of variable and constant.



(c) Argument of `eval` in Figure 4.1.



(d) Argument of `exec` in Figure 4.1.

FIGURE 7.2: Examples of template trees.

Definition 7.2.1 (Template tree) *A template tree is an acyclic, connected, directed graph* $(\mathcal{N}, \mathcal{E})$ *where*

- *a node* $n \in \mathcal{N}$ *represents a string constant, a symbolic variable, an operator, or an alternative, and*
- *an edge* $e \in \mathcal{E}$ *represents a nesting relationship between two nodes.*

Figure 7.2 shows several examples of template trees:

- Example (a) represents a value known to be a string constant `"ls -l"`. The template tree consist of a single node labeled with this string.
- In example (b), the analysis knows that the value is the result of concatenating the value of a symbolic variable $*cmd* and the string constant `" -l"`. The root node of the template tree is a concatenation operator, which has two child nodes: a symbolic variable node and a string constant node.
- Example (c) shows the tree that the analysis extracts for the values that may be passed to `eval` at line 10 of Figure 4.1. Because the value

Example	Templates
(a)	$t_{a1} = [$`"ls -l"`$]$
(b)	$t_{b1} = [\$cmd,$`" -l"`$]$
(c)	$t_{c1} = [$`"messages.backup_pics"`$]$
	$t_{c2} = [$`"messages.backup_other"`$]$
(d)	$t_{d1} = [$`"cp "`$,\$name,$`"."`$,\$ext,,$ `" ~/.localBackup"`$]$

FIGURE 7.3: Evaluated templates for the examples in Figure 7.2.

depends on the condition checked at line 9, the tree has an alternative node with children that represent the two possible string values.

- Finally, example (d) is the tree extracted for the value passed to `exec` at line 7 of Figure 4.1. This tree contains several operation nodes that represent the push operations and the string concatenation that are used to construct the string value, as well as several symbolic variable nodes and string constant nodes.

To extract such templates trees automatically, we use a data flow analysis [ASU86; NNH05], which propagates template trees through the program. Starting at a call site of an injection API with an empty tree, the analysis applies the following transfer functions:

- *Constants.* Reading a string constant yields a node that represents the value of the constant.
- *Variables.* A read of a local variable or a function parameter yields a node that represents a symbolic variable.
- *Operations.* Applying an operation, such as concatenating two strings with +, yields a tree where the root node represents the operator and its children represent the operands.
- *Calls.* A call of a function yields a tree where the root node represents the called function and its children represent the base object and arguments of the call.

- *Assignments.* An assignment of the form *lhs* = *rhs* transforms the current tree by replacing any occurrence of the symbolic variable that corresponds to *lhs* by the tree that results from applying the transition function to *rhs*.

Whenever the backward control flow merges, the analysis merges the two template trees of the merged branches. The merge operation inserts an alternative node that has the two merged trees as its children. To avoid duplicating subtrees with identical information, the analysis traverses the two given trees t_1 and t_2 to find the smallest pair of subtrees t_1' and t_2' that contain all differences between t_1 and t_2, and then inserts the alternative node as the parent of t_1' and t_2'.

Template tree construction example. For example, consider the call site of `eval` at line 10 of Figure 4.1. Starting from an empty tree, the analysis replaces the empty tree with a tree that represents the string concatenation at line 10. One child of this tree is a variable node that represents the variable `kind`, which has an unknown value at this point. Then, the analysis reasons backwards and follows the two control flow paths that assign `"pics"` and `"other"` to the variable `kind`, respectively. For each path, the analysis updates the respective tree by replacing the variable node for `kind` with the now known string constant. Finally, the variable reaches the merge point of the backward control flow and merges the two trees by inserting an alternative node, which yields the tree in Figure 7.2c.

7.2.2 *Evaluating Template Trees*

Based on the template trees extracted by the backward data flow analysis, the second step of the static analysis is to evaluate the tree for each call site of an injection API. The result of this evaluation process is a set of templates:

Definition 7.2.2 (Template) *A template is a sequence* $t = [c_1, \ldots, c_k]$ *where each* c_i *represents either a constant string or an unknown value (hole).*

For example, the template trees in Figure 7.2 are evaluated to the templates in Figure 7.3. To obtain the templates for a given tree, the analysis iteratively evaluates subtrees in a bottom-up way until reaching the root node. The evaluation replaces operation nodes that have a known semantics with the result of the operation. Our implementation currently models the semantics of string concatenation, `Array.push`, `Array.join`, and

`String.replace` where the arguments are constant strings. These operations cover most templates trees that the analysis extracts from real-world JavaScript code (Section 7.5.1). For alternative nodes, the evaluation considers both cases separately, duplicating the number of template trees that result from the evaluation.

Finally, the analysis transforms each evaluated tree into a template by joining continuous sequences of characters into constant strings and by representing all symbolic values and unknown operations between these constants as unknown values.

7.2.3 *Identifying Statically Safe Calls*

After evaluating template trees, the analysis knows for each call site of an injection API the set of templates that represent the string values passed to the call. If all templates for a particular call site are constant strings, i.e., there are no unknown parts in the template, then the analysis concludes that the call site is statically safe. For such statically safe call sites, no runtime checking is required. In contrast, the analysis cannot statically ensure the absence of injections if the templates contain unknown values. In this case, checking is deferred to runtime, as explained in Section 7.3.

For our running example, the analysis determines that the `eval` call site at line 10 of Figure 4.1 is statically safe because both possible values passed to the function are known. In contrast, parts of the strings that may be passed to `exec` at line 7 are unknown and therefore the check whether an injection happens is deferred to runtime.

7.3 DYNAMIC ENFORCEMENT

For call sites where the values passed to the injection API cannot be statically determined, we provide a dynamic enforcement mechanism. The goal of this mechanism is to reject values found to be dangerous according to a policy. Intuitively, we want to prevent values that expand the template computed for the call site in a way that is likely to be unforeseen by the developer. Our approach achieves this goal in two steps:

1. Before executing the module, the approach transforms the statically extracted set of templates for a call site into a set of partial abstract syntax trees (PAST) that represents the expected structure of benign values. The trees are partial because the unknown parts of the template are represented as unknown subtrees.

2. While executing the module, the approach parses the runtime value passed to an injection API into an AST and compares the PASTs from step 1 against the AST. The runtime mechanism enforces a policy that ensures that the runtime AST is (i) derivable from at least one of the PASTs by expanding the unknown subtrees and (ii) these expansions remain within an allowed subset of all possible AST nodes.

The following two subsections present the two steps of the dynamic enforcement mechanism in detail.

7.3.1 *Synthesizing a Tree-based Policy*

The goal of the first step is to synthesize for each call site a set of trees that represents the benign values that may be passed to the injection API. Formally, we define these trees as follows:

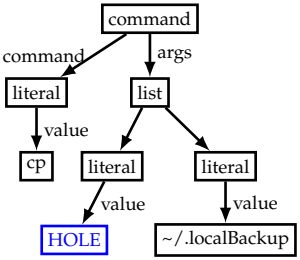
Definition 7.3.1 (Partial AST) *The partial AST (PAST) for a template of an injection API call site is an acyclic, connected, directed graph* $(\mathcal{N}, \mathcal{E})$ *where*

- $\mathcal{N}_{sub} \subseteq \mathcal{N}$ *is a set of nodes that each represent a subtree of which only the root node* $n_{sub} \in \mathcal{N}_{sub}$ *is known, and*
- $(\mathcal{N}, \mathcal{E})$ *is a tree that can be expanded into a valid AST of the language accepted by the API.*

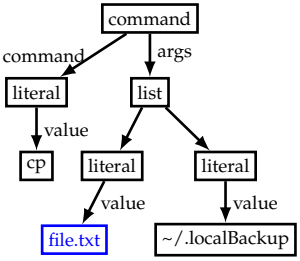
For example, Figure 7.4a shows the PAST for the template t_{d1} from Figure 7.3. For this partial tree, $\mathcal{N}_{sub} = \{\text{HOLE}\}$, i.e., the hole node can be further expanded, but all the other nodes are fixed.

To synthesize the PAST for a template, the approach performs the following steps. At first, it instantiates the template by filling its unknown parts with simple string values known to be benign. The set of *known benign values* must be defined only once for each injection API. Figure 7.5 shows the set of values we use for `exec` and `eval`, respectively. The approach exhaustively tries all possible assignments of these values to the unknown parts of a template. Then, each of the resulting strings is given to a parser of the language, e.g., a JavaScript or Bash parser. If and only if the string is accepted as a legal member of the language, then the approach stores the resulting AST into a set of *legal example ASTs*.

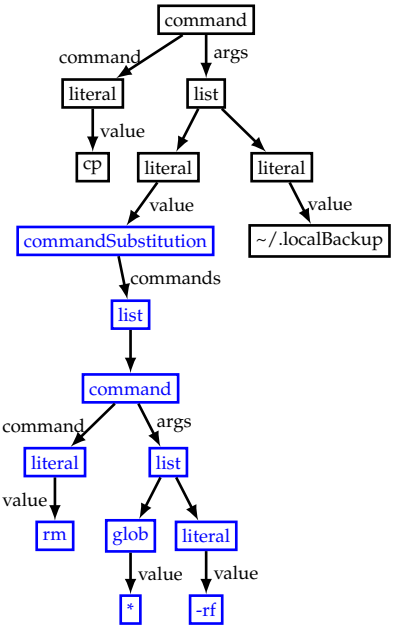
Given the legal example ASTs for a template, the next step is to merge all of them into a single PAST. To this end, the approach identifies the least common nodes of all ASTs, i.e., nodes that are shared by all ASTs but that



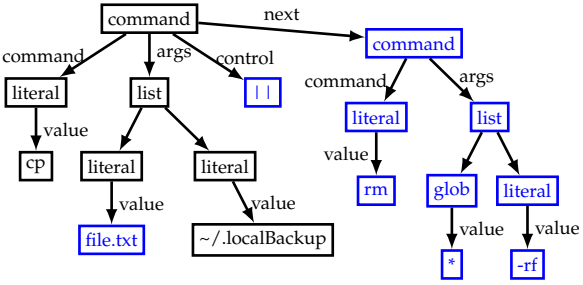
(a) Partial AST for the template in Figure 7.3.



(b) AST accepted by the policy derived from the partial AST.



(c) AST rejected by the policy derived from the partial AST.



(d) AST rejected by the policy derived from the partial AST.

FIGURE 7.4: A partial AST and three ASTs compared against it. The blue nodes are holes and runtime values filled into the holes at runtime.

API	Language	Known benign values
`exec`	Bash	`"./file.txt"`, `"ls"`
`eval`	JavaScript	`x`, `y`, `"x"`, `x.p`, `{x:23}`, `23`

FIGURE 7.5: Known benign values used to synthesize PASTs.

have a subtree that differs across the ASTs. At first, the given ASTs are aligned by their root nodes, which must match because all ASTs belong to the same language. Then, the approach simultaneously traverses all ASTs in a depth-first manner and searches for nodes n_{lc} with children that differ across at least two of the ASTs. Each such node n_{lc} is a least common node. Finally, the approach creates a single PAST that contains the common parts of all ASTs and where the least common nodes remain unexpanded and form the set $\mathcal{N}_{sub}$ (Definition 7.3.1). Note that $\mathcal{N}_{sub}$ is effectively an under-approximation of the possible valid inputs, given that we construct it using a small number of known benign inputs. However, in practice we do not observe any downsides to this approach, as discussed in Section 7.5.

Policy synthesizing example. For example, for the template t_{d1} and the known benign inputs for Bash in Figure 7.5, the first argument passed to `cp` will be expanded to the following values: `./file.txt.ls`, `ls.ls`, `./file.txt../file.txt` and `ls../file.txt`. All these values are valid literals according to the Bash grammar, i.e., we obtain four legal example ASTs. By merging these ASTs, the approach obtains the PAST in Figure 7.4a because the only variations observed across the four ASTs are in the value of the literal.

7.3.2 *Checking Runtime Values Against the Policy*

The set of PASTs synthesized for a call site is the basis of a policy that our mechanism enforces for each string passed at the call site. We implement this enforcement by rewriting the underlying JavaScript code at the call site. When a runtime value reaches the rewritten call site, then the runtime mechanism parses it into an AST and compares it with the PASTs of the call site. During this comparison, the policy enforces two properties:

- **P1**: The runtime value must be a syntactically valid expansion of any of the available PASTs. Such an expansion assigns to each node

$n_{sub} \in \mathcal{N}_{sub}$ a subtree so that the resulting tree (i) is legal according to the language and (ii) structurally matches the runtime value's AST.

- **P2**: The expansion of a node n_{sub} of the PAST is restricted to contain only AST nodes from a pre-defined set of *safe node types*. The set of safe node types is defined once per language, i.e., it is independent of the specific call site and its PASTs. For shell commands passed to `exec`, we consider only nodes that represent literals as safe. For JavaScript code passed to `eval`, we allow all AST node types that occur in JSON code, i.e., literals, identifiers, properties, array expressions, object expressions, member expressions, and expression statements. The rationale for choosing safe node types is to prevent an attacker from injecting code that has side effects. With the above safe node types, an attacker can neither call or define functions, nor update the values of properties or variables. As noted in previous work [RL12], such a restrictive mechanism may cause false positives, which we find to be manageable in practice though.

Policy checking example. To illustrate these properties, suppose that the three example inputs in Figure 7.6 are given to the `backupFile` function in Figure 4.1. Input 1 uses the function as expected by the developer. In contrast, inputs 2 and 3 exploit the vulnerability in the call to `exec` by passing data that will cause an additional command to be executed. Figure 7.4 shows the PAST derived (only one because there is only one template available for this call site) for the vulnerable call site and the ASTs of the three example inputs. Input 1 fulfills both P1 and P2 and the value is accepted. In contrast, the policy rejects input 2 because it does not fulfill P1. The reason is that the AST of the input (Figure 7.4d) does not structurally match the PAST. Likewise, the policy rejects input 3 because it fulfills P1 but not P2. The reason for not fulfilling P2 is that the expanded subtree (i.e., the highlighted nodes in Figure 7.4c) contain nodes that are not in the set of safe node types.

To summarize, the enforced policy can be formalized as follows:

Definition 7.3.2 (Security Policy) *Given a runtime value v, a set of PASTs $\mathcal{T}$, and a set $\mathcal{N}_{safe}$ of safe node types, v is* rejected *unless there exists an expansion t' of some $t \in \mathcal{T}$, where*

- *t' is isomorphic to the AST of v, and*

ID	name	ext	Property	
			P1	P2
1	file	txt	✓	✓
2	file	txt || rm * -rf	✗	–
3	file	$(rm * -rf)	✓	✗

FIGURE 7.6: Inputs compared against the partial AST in Figure 7.4a.

- *let $\mathcal{N}_{input}$ be the set of nodes that belong to a subtree in the AST of v that matches a node in $\mathcal{N}_{sub} \in t$, then the node type of all $n \in \mathcal{N}_{input}$ is in $\mathcal{N}_{safe}$.*

Our runtime enforcement approach can be applied to any kind of injection API that expects string values specified by a context-free grammar. The effectiveness of the enforcement depends on two language-specific ingredients: the set of benign example inputs and the set of safe AST node types. Given that we are primarily interested in `eval` and `exec` sinks, we have created these ingredients for JavaScript and Bash, and Section 7.5.2 shows both to be effective for real-world NODE.JS code.

7.4 IMPLEMENTATION

Static analysis: We implement the static analysis in Java, building upon the Google Closure Compiler[3]. The analysis is an intraprocedural, backward data flow analysis, as described in Section 7.2.1. The states propagated along the control flow edges are sets of template trees and the join operation is the union of these sets. To handle loops and recursion, the static data flow analysis limits the number of times a statement is revisited while computing a particular data flow fact to ten. When applying the static analysis to a module, we impose a one minute timeout per module. Considering the deployment strategy we propose for SYNODE later in this section, we believe that an analysis that takes longer would be of little practical use. We show in the evaluation that the cases in which the timeout expires are rare and therefore for these cases, SYNODE alerts the user that a manual inspection is needed. As described in Section 7.2.2, after finishing the data flow analysis of a module, the implementation transforms the col-

3 https://developers.google.com/closure/

lected template trees into templates. Lastly, the analysis writes the set of templates for each call site into a file to be used by the dynamic analysis.

Runtime analysis: We implement the dynamic analysis in JavaScript. Before executing the module, the analysis pre-computes the PASTs for each call site based on the templates gathered by the static analysis. While executing a module, the analysis intercepts all calls to `exec` and `eval` and extracts the strings passed to these function to be checked against our policy. To parse strings given to `exec` and `eval`, we build upon the esprima[4] and shell-parse[5] modules.

Automatic deployment: As shown by our injections study (Section 4.3), the practical benefits of a technique to prevent injection attacks depend on how seamlessly the technique can be deployed. A particular challenge is how to apply a mitigation technique to code written by third parties that may not be willing to modify their code. To make the deployment of Synode as easy as possible without relying on the cooperation of third-party code providers, we advocate an approach in which a module developer or a system administrator adds a post-installation script[6] to the application packaged as an npm module.

The script runs on each explicitly declared third-party dependent module and, if necessary, performs the code rewriting step that adds dynamic enforcement at each statically unsafe call site of an injection API. As a result, our technique to prevent injection attacks can be deployed with very little effort and without requiring any knowledge about third-party code.

7.5 Evaluation

We evaluate our mitigation technique by applying it to all 235,850 Node.js modules. To avoid analyzing modules without any injection call sites, we filter modules by searching for call sites of these methods and include all 15,604 modules with at least one such call site in our evaluation. We apply our static analysis for each module separately to decide whether the sink call sites are statically safe or runtime protection is needed for that module. Since evaluating the runtime mechanism requires inputs that exercise the modules, we consider a subset of the modules, with known vulnerabilities, found by others or by us during the study (Section 4.3).

4 http://esprima.org/
5 https://www.npmjs.com/package/shell-parse
6 https://docs.npmjs.com/misc/scripts

Kind of template tree	Call sites	
	`exec`	`eval`
Evaluates to constant string without holes	31.05%	39.29%
Holes due to symbolic variables only	49.02%	34.52%
Holes due to unsupported operations	19.93%	26.19%

FIGURE 7.7: Template trees extracted by the static analysis.

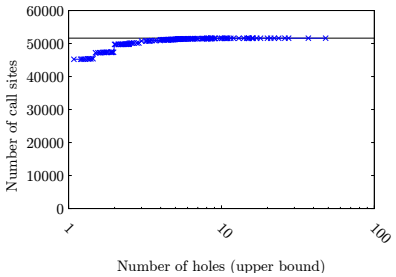
We perform all our measurements on a Lenovo ThinkPad T440s laptop with an Intel Core i7 CPU (2.10GHz) and 12 GB of memory, running Ubunu 14.04.

7.5.1 *Static Analysis*

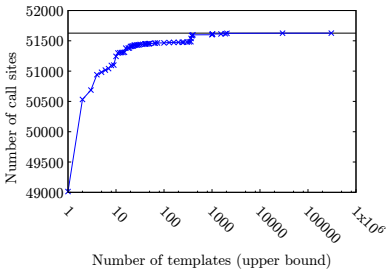
Statically safe call sites: The static analysis finds 18,924 of all 51,627 call sites (36.66%) of injection APIs to be statically safe. That is, the values that are possibly passed to each of these call sites are statically known, and an attacker cannot modify them. To further illustrate this point, Figure 7.7 shows to what extent the analysis can evaluate trees into templates. For 31.05% and 39.29% of all call sites of `exec` and `eval`, respectively, the template tree contains only constant nodes, operators supported by the analysis, and alternative nodes, which yield constant strings after evaluating the tree.

The remaining template trees also contain symbolic variable nodes. Most of these trees (49.02% and 34.52%) are fully evaluated by the analysis, i.e., they contain no unsupported operators. It is important to note that the static analysis may provide a useful template even if the template tree contains an unsupported operation. The reason is that the other nodes in the tree often provide enough context around the unknown part created by the unsupported operation.

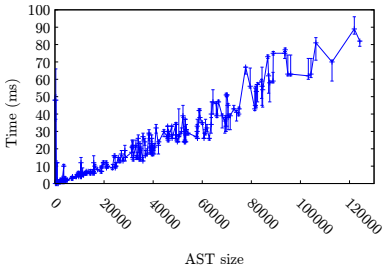
Context encoded in templates: To better understand the templates extracted by the static analysis, we measure how much context about the passed string the static analysis extracts. First, we measure for each call site how many known characters are present per template, on average. The majority of call sites contain at least 10 known characters and for 10,967 call sites (21.24%), there is no known character, i.e., our approach relies entirely on dynamic information. Second, we measure how many unknown parts



(a) CDF for the average number of holes per call site. Note the logarithmic horizontal axis.



(b) CDF for the number of inferred templates per call site. Note the logarithmic horizontal axis.



(c) Overhead of runtime checks depending on input size.

FIGURE 7.8: Details on static analysis and overhead of runtime checks.

the extracted templates contain. As shown in Figure 7.8a, the templates for the vast majority of call sites has at most one hole, and very few templates contain more than five holes.

The main reason for templates with a relatively large number of holes is that the string passed to injection API is constructed in a loop that appends unknown values to the final string. The analysis unrolls such loops a finite number of times, creating a relatively large number of unknown parts.

Third, we measure how many templates the analysis extracts per call site. Because different executed paths may cause different string values to be passed at a particular call site of an injection API, the analysis may yield multiple templates for a single call site. Figure 7.8b shows that for most call sites, a single template is extracted.

Reasons for imprecision: To better understand the reasons for imprecision of the static analysis, we measure how frequent particular kinds of nodes

Type	Benchmark Module	Injection vector	Inputs benign	Inputs malicious	False negatives	False positives	Average overhead (ms)
Advisories (Sect. 4.2)	gm	I	1	2	0	0	0.41
	libnotify	I	4	2	0	1	0.19
	codem-transcode	N	1	4	0	0	0.80
	printer	I	1	4	0	0	0.28
Reported by us (Sect. 4.3.4)	mixin-pro	I	2	4	0	0	0.16
	modulify	I	1	2	0	1	0.04
	mol-proto	I	1	2	0	1	0.07
	mongoosify	I	1	2	0	0	0.04
	mobile-icon-resizer	FS	1	5	0	0	0.39
	m-log	I	11	1	0	0	0.05
	mongo-parse	I	1	2	0	0	0.11
	mongoosemask	I	1	1	0	0	0.04
	mongui	N	1	2	0	0	0.05
	mongo-edit	N	1	1	0	0	0.04
	mock2easy	N	1	2	0	0	0.03
Case study (Sect. 4.3.5)	growl	I	1	2	0	0	2.72
	autolint	FS	4	4	0	0	1.59
	mqtt-growl	N	1	2	0	0	3.19
	chook-growl-reporter	I	1	1	0	0	1.60
	bungle	FS	14	4	0	0	1.99
Other exec (Sect. 4.3.2)	fish	I	1	4	0	0	0.21
	git2json	I	1	4	0	1	0.37
	kerb_request	I	3	4	0	0	0.25
	keepass-dmenu	CL	1	4	0	1	0.52
	Total		56	65	0	5	
	Average						0.74

FIGURE 7.9: Results for runtime enforcement. Used injection vectors: module's interface (I), network (N), file system (FS), command line (CL).

```
1 var keys = Object.keys(dmenuOpts);
2 var dmenuArgs = keys.map(function (flag) {
3   return '-' + flag + ' "' + dmenuOpts[flag] + '"';
4 }).join(' ');
5 var cmd = 'echo | dmenu -p "Password:" ' + dmenuArgs;
6 exec(cmd);
```

FIGURE 7.10: Example of a false positive.

in template trees are. We find that 17.48% of all call sites have a template tree with at least one node that represent a function parameter. This result suggests that an inter-procedural static analysis might collect even more context than our current analysis. To check whether the static analysis may miss some sanitization-related operations, we measure how many of the nodes correspond to string operations that are not modelled by the analysis and to calls of functions whose name contains "escape", "quote", or "sanitize". We find that these nodes appear at only 3.03% of all call sites. The low prevalence of such nodes, reiterates the observation we made during our study in Chapter 4: An npm module that uses sanitization when calling an injection API is the exception, rather than the rule.

Analysis running time: Our analysis successfully completes for 96.27% of the 15,604 modules without hitting the one-minute timeout after which we stop the analysis of a module. The average analysis time for these modules is 4.38 seconds, showing the ability of our approach to analyze real-world code at a very low cost.

We conclude from these results that the static analysis is effective for the large majority of call sites of injection APIs. Either the analysis successfully shows a call site to receive only statically known values, or it finds enough context to yield a meaningful security policy to be checked at runtime. This finding confirms our design decision to use a scalable, intra-procedural analysis. The main reason why this approach works well is because most strings passed to injection APIs are constructed locally and without any input-dependent path decisions.

7.5.2 *Runtime Mechanism*

For evaluating the runtime mechanism we consider a set of 24 vulnerable modules listed in Figure 7.9. The set includes modules reported as vulnerable on the Node Security Platform, modules with vulnerabilities found

during our study in Chapter 4, and clients of known vulnerable modules. We exercise each module both with benign and with malicious inputs that propagate to the call sites of injection APIs.[7] As benign inputs, we derive example usages from the documentation of a module. As malicious inputs, we manually craft payloads that accomplish a specific goal. The goal for `eval` is to add a particular property to the globally available `console` object. For `exec`, the goal is to create a file in the file system. Figure 7.9 lists the modules and the type of injection vector we use for the malicious inputs. "(I)nterface" means that we call the module via one of its exported APIs, "(N)etwork" means that we pass data to the module via a network request, "(F)ile system" means that the module reads input data from a file, and "(C)ommand line" means that we pass data as a command line argument to the module. In total, we use 56 benign inputs and 65 malicious inputs.

False positives: Across the 56 benign inputs, we observe five false positives, i.e., a false positive rate of only 8.92%. Three false positives are caused by limitations of our static analysis. For example, Figure 7.10 contains code that constructs a command passed to `exec` by transforming an array `keys` of strings using `Array.map`. Because our static analysis does not model `Array.map`, it assumes that the second part of `cmd` is unknown, leading to a PAST with a single unknown subtree. Our runtime policy allows filling this subtree with only a single argument, and therefore rejects benign values of `dmenuArgs` that contain two arguments. Further improvements of our static analysis, e.g., by modeling built-in functions, will reduce this kind of false positive.

The remaining two false positives are caused by the set of safe node types in our runtime mechanism. For example, the mol-proto module uses `eval` to let a user define arbitrary function bodies, which may include AST nodes beyond our set of safe node types. Arguably, such code should be refactored for enhanced security. Alternatively, if a user of a module trust that module, she can whitelist either specific call site of the injection API or the entire module.

Overall, we conclude that the approach is effective at preventing injections while having a false positive rate that is reasonably low, in particular for a fully-automated technique.

False negatives: Synode prevents all attempted injections during our evaluation, i.e., there are not false negatives. In general, however, there are

7 The modules and inputs are available as benchmarks for future research: https://github.com/sola-da/Synode.

multiple reasons that might cause false negatives. First, our static analysis fails to identify highly dynamic sink calls:

```
global["ev"+"al"](userInput);
```

Because the code to construct call targets can be arbitrarily complex, no static analysis can guarantee to detect all sink calls. As Synode targets code that is vulnerable by accident, and not malicious on purpose, we consider the problem of hidden sink calls to be negligible in practice. Second, Synode prevents the addition of new commands to the templates, but it does not defend against data only attacks. For example, sometimes it is insufficient to ensure that the input is a literal:

```
exec("rm " + userInput)
```

Computational cost: Our static analysis identifies a total of 1,560 templates for the injection APIs in the considered modules. For each of them, we construct a PAST with a median computation time of 2 milliseconds per module. We note that for some modules this number is significantly higher due to our simple PAST construction algorithm and due to the high number of templates per module.

The last column of Figure 7.9 shows the average runtime overhead per call of an injection API that is imposed by the runtime mechanism (in milliseconds). We report absolute times because the absolute overhead is more meaningful than normalizing it by a fairly arbitrary workload. Our enforcement mechanism costs 0.74 milliseconds per call, on average over 100 runs of the modules using all the inputs. This result demonstrates that the overhead of enforcement is generally negligible in practice.

To demonstrate the scalability of our runtime enforcement, we consider input data of different size and complexity and pass it to the injection APIs. Here, we focus on `eval` call sites from Figure 7.9 only. As inputs, we use a diverse sample of 200 JavaScript programs taken from a corpus of real-world code[8]. For every call to `eval`, we pass all 200 JavaScript programs 100 times each and measure the variance in enforcement times. Figure 7.8c shows the enforcement time, in milliseconds, depending on the size of the JavaScript program, measured as the number of AST nodes. For each input size, the figure shows the 25% percentile, the median value, and the 75% percentile. We find that the enforcement time scales linearly. The reason is that all steps of the runtime enforcement, i.e., parsing the input, matching the AST with the PASTs, and checking whether nodes are on a whitelist, are of linear complexity.

8 http://learnbigcode.github.io/datasets/

7.6 CONCLUSIONS

In this chapter we present SYNODE, an automated technique for mitigating injection vulnerabilities in NODE.JS applications. At the same time, the approach effectively prevents a range of attacks while causing very few false positives and while imposing sub-millisecond overheads. To aid with its adoption, our technique requires virtually no involvement on the part of the developer. Instead, SYNODE can be deployed automatically as part of module installation.

8

FULL-STACK INFORMATION FLOW ANALYSIS

Information flow analysis is a powerful security program analysis that can be used to secure third-party JavaScript code and, hence, mitigate some of the security risks incurred by excessive code reuse (see particularity P_2 in the introduction). This chapter studies the tradeoff caused by considering different types of implicit flows in an information flow analysis. It is full-stack in the sense that in our evaluation we consider both client-side and server-side benchmarks. This chapter shares material with the corresponding publication [Sta+19].

8.1 MOTIVATION

While some language features JavaScript, such as dynamism and flexibility, explain its popularity, the lack of other features, such as language-level protection and isolation mechanisms, open up a wide range of integrity, availability, and confidentiality vulnerabilities [Joho8]. As a result, securing JavaScript applications has become a key challenge for web application security. Unfortunately, existing browser-level mechanisms, such as the same-origin policy or the content security policy, are coarse-grained, falling short to distinguish between secure and insecure manipulation of data by scripts. Furthermore, server-side applications lack such isolation mechanisms completely, allowing an attacker, e.g., to inject and execute arbitrary code that interacts with the operating system through powerful APIs as we show in Chapter 4.

An appealing approach to securing JavaScript applications is information flow analysis. This approach tracks the flow of information from sources to sinks in order to enforce application-level security policies. It can ensure both integrity, by preventing information from untrusted sources to reach trusted sinks, and confidentiality, by preventing information from secret sources to reach public sinks. For example, information flow analysis can check that no attacker-controlled data is evaluated as executable code or that secret user data is not sent to the network. Because the dynamic nature of JavaScript hinders precise static analysis, dynamic information flow analysis has received significant attention by researchers [AF12; Bau+15;

```
1  // variable passwd is sensitive
2  var gotIt = false;
3  var paddedPasswd = "xx" + passwd;
4  var knownPasswd = null;
5  if (paddedPasswd === "xxtopSecret") {
6    gotIt = true;
7    knownPasswd = passwd;
8  }
9  // function sink is insensitive
10 sink(gotIt);
```

FIGURE 8.1: Program leaking the password to the network.

Bic+17; CN15; DG+12; Hed+14; RJLS10; Vog+07]. The basic idea of dynamic information flow analysis is to attach security labels, e.g., secret (untrusted) and public (trusted), to runtime values and to propagate these labels during program execution. To simplify the presentation, we assume to have two security labels, and we say that a value is *sensitive* if its label is secret or untrusted; otherwise, we say that a value is *insensitive*.

At the language level, a program may propagate information via two kinds of information flows:[1] *Explicit flows* [DD77] occur whenever sensitive information is passed by an assignment statement or into a sink. *Implicit flows* [DD77] arise via control-flow structures of programs, e.g., conditionals and loops, when the flow of control depends on a sensitive value. For a dynamic information flow analysis, implicit flows can be further classified into flows that happen because a particular branch is executed, so-called *observable implicit flows* [BSS17], and flows that happen because a particular branch is not executed, so called *hidden implicit flows* [BSS17].

Figure 8.1 illustrates the different kinds of flows with a simple JavaScript-like program that leaks sensitive information. The program has a variable `passwd`, which is marked initially as a sensitive source at line 1. Using this variable in an operation that creates a new value, e.g., in line 3, is an explicit flow. Consider the case where the password is "topSecret", i.e., the conditional at line 5 evaluates to `true`, and line 6 sets `gotIt` to `true`. At line 10, the `gotIt` variable is sent to the network through the function `sink()`, which is considered to be an insensitive sink. The flow from the password to `gotIt` is an observable implicit flow because a sensitive value determines that `gotIt` gets written. Now, consider the case where `passwd` is "abc". The branch at line 5 is not taken and the `gotIt` variable remains `false`. Sending this information to the network reveals that the password is different from "topSecret". This flow is a hidden implicit flow because a sensitive value determines that `gotIt` does not get written.

1 There are other kinds of flows, such as timing and cache side-channels, which we ignore here.

Ideally, an information flow analysis should consider all three kinds of flows. In fact, there exists a large body of work on static, dynamic, hybrid, and multi-execution techniques to prevent explicit and implicit flows. However, so far these tools have seen little use in practice, despite the strong security guarantees that they provide. In contrast, a lightweight form of information flow analysis called *taint analysis* is widely used in computer security [SAB10]. Taint analysis is a pure data dependency analysis that only tracks explicit flows, ignoring any control flow dependencies.

The question which kinds of flows to consider is a tradeoff between costs and benefits. On the cost side, considering more flows increases false positives [Kin+08]. A false positive here means that a secure execution is conservatively blocked by an overly restrictive enforcement mechanism. A common reason is that a value gets labeled as sensitive even though it does not actually contain information that is security-relevant in practice. This problem, sometimes referred to as *label creep* [Den82; SM03], reduces the permissiveness of information flow monitoring, because the monitor will prematurely stop a program to prevent a value with an overly sensitive label from reaching a sink. Another cost of considering more kinds of flows is an increase in runtime overhead. On the benefit side, considering more flows increases the ability to find security vulnerabilities and data leakages, i.e., the level of trust one obtains from the analysis. For example, an analysis that considers only explicit flows will miss any leakage of sensitive data that involves an implicit flow. Unfortunately, despite the large volume of research on information flow analysis, there is very little empirical evidence on the importance of the different kinds of flows in real applications. Because of this lack of knowledge, potential users of information flow analyses cannot make an informed decision about what kind of analysis to use.

To better understand the tradeoff between costs and benefits of using a dynamic information flow analysis, this chapter presents an empirical study of information flows in real-world JavaScript code. Our overall goal is to better understand the costs and benefits of dynamically analyzing explicit, observable implicit, and hidden implicit flows. Specifically, we are interested in how prevalent different kinds of flows are, what kinds of security problems can(not) be detected when considering subsets of flows, and what costs considering all flows imposes. To address these questions, we study 56 real-world JavaScript programs in various application domains with a diverse set of security policies. The study considers integrity problems, specifically code injection vulnerabilities and denial of service vulner-

abilities caused by an algorithmic complexity problem, and confidentiality problems, specifically leakages of uninitialized memory, browser fingerprinting and history sniffing. Each studied program has at least one real-world security problem that information flow analysis can detect.

Our study is enabled by a novel methodology that combines state-of-the-art dynamic information flow analysis [AF10; HS12; Hed+14] and program rewriting [BHS12] with a set of novel *security metrics*. We implement the methodology in a dynamic information flow analysis built on top of Jalangi [Sen+13]. The implementation draws on a sound analysis for a simple core of JavaScript. The formalization relates the security metrics to semantic security conditions for taint tracking [Sch+16], observable tracking [BSS17] and information flow monitoring [GM82].

The findings of our study include:

1. All three kinds of flows occur locally in real-life applications, i.e., an analysis that ignores some of them risks to miss violations of the information flow policy. Explicit flows are by far the most prevalent, and only five benchmarks contain hidden implicit flows (Section 8.4.1).

2. An analysis that considers explicit and observable implicit flows, but ignores hidden implicit flows, detects all vulnerabilities in our benchmarks. For most applications it is even sufficient to track explicit flows only, while for some client-side, privacy-related applications one must also consider observable implicit flows (Section 8.4.2).

3. Tracking hidden implicit flows causes an analysis to prematurely terminate various executions. Furthermore, we find that different monitoring strategies proposed in the literature vary significantly in their permissiveness. (Section 8.4.3).

4. The amount of data labeled as sensitive steadily increases during the execution of most benchmarks, confirming the label creep problem. An analysis that considers implicit flows increases the label creep by over 40% compared to an analysis that considers only explicit flows (Section 8.4.4).

5. The analysis overhead caused by considering implicit flows is significant: Ignoring implicit flows saves the effort of tracking runtime operations by a factor of 2.5 times (Section 8.4.5).

Prior work (discussed in Section 10.8) studies false positives caused by static analysis of implicit flows [Kin+08; RSL09] and the semantic strength

of flows [MP09]. Jang et al. [RJLS10] conduct a large-scale empirical study showing that several popular web sites use information flows to exfiltrate data about users' behavior. Kang et al. [Kan+11] combine dynamic taint analysis with targeted implicit flow analysis, demonstrating the importance of tracking implicit flows for trusted programs. However, to the best of our knowledge, no existing work analyzes the cost-benefit tradeoff of considering different types of flows in realistic JavaScript programs.

In summary, this chapter contributes the following:

- We are the first to empirically study the prevalence of explicit, observable implicit, and hidden implicit flows in real-world applications against integrity, availability, and confidentiality policies.
- We present a methodology and its implementation, which enables the study, and we provide a formal basis for empirically studying information flows (Section 8.3).
- Through realistic case studies and security policies, we provide empirical evidence that sheds light on the cost-benefit tradeoff of information analysis and that outlines directions for future work (Section 8.4).

We implement our dynamic information flow analysis in a tool called IFLOW which we make available, together with all the benchmarks and policies used for the study, to support future evaluations of information flow tools for JavaScript.[2]

8.2 BENCHMARKS AND SECURITY POLICIES

Our study is based on 56 client-side and server-side JavaScript applications, which suffer from four classes of vulnerabilities. These applications are subject to attacks that have been independently discovered by existing work, including integrity, availability, and confidentiality attacks. For every application, we define realistic security policies expressed as information flow policies. Table 8.1 shows the applications, along with their security policies, and size measured in lines of code. The benchmarks vary in size from tens of lines of code to tens of thousands. We further explain the policies below. For each application we either create or reuse a set of inputs that trigger the attack and other inputs to increase the coverage of different behaviors.

2 https://new-iflow.herokuapp.com/download-iflow.html

Type	ID	Library	Policy	LoC	SBC	Upgs
Injection vulnerabilities	1	fish	module → eval and exec	69	1	0
	2	growl	module → eval and exec	270	1	0
	3	gm	module → eval and exec	1,614	1	0
	4	libnotify	module → eval and exec	54	1	0
	5	mixin-pro	module → eval and exec	168	1	0
	6	modulify	module → eval and exec	2,410	1	0
	7	mol-proto	module → eval and exec	1,696	1	0
	8	mongoosify	module → eval and exec	160	0	1
	9	m-log	module → eval and exec	243	1	0
	10	mobile-icon-resizer	file system API → eval and exec	410	1	0
	11	mongo-parse	module → eval and exec	506	1	0
	12	mongoosemask	module → eval and exec	12,750	0.78	28
	13	mongui	HTTP API → eval and exec	1,539	0.44	0
	14	mongo-edit	HTTP API → eval and exec	577	0	0
	15	mock2easy	HTTP API → eval and exec	1,217	0.07	3
	16	chook-growl-reporter	module → eval and exec	243	1	0
	17	git2json	module → eval and exec	434	1	0
	18	kerb_request	module → eval and exec	67	1	0
	19	printer	module → eval and exec	139	1	0
ReDoS vulnerabilities	20	debug	module → regex matching	360	1	0
	21	mime	module → regex matching	108	1	0
	22	tough-cookie	module → regex matching	1,145	1	0
	23	fresh	module → regex matching	59	0.5	0
	24	forwarded	module → regex matching	30	0	0
	25	underscore.string	module → regex matching	1,779	1	0
	26	ua-parser-js	module → regex matching	584	0.50	6
	27	parsejson	module → regex matching	46	1	0
	28	useragent	module → regex matching	6,827	1	0
	29	no-case	module → regex matching	33	1	0
	30	content-type-parser	module → regex matching	221	1	0
	31	timespan	module → regex matching	577	0.20	4
	32	string	module → regex matching	2,001	1	0
	33	content	module → regex matching	125	0.42	0
	34	slug	module → regex matching	375	0.5	2
	35	htmlparser	module → regex matching	2,155	0.65	5
	36	charset	module → regex matching	49	0.5	0
	37	mobile-detect	module → regex matching	612	1	0
	38	ismobilejs	module → regex matching	935	0.33	1
	39	dns-sync	module → regex matching	76	1	0
Buffer vulns.	40	ip	buffer reading → module	325	0.76	0
	41	concat-stream	buffer reading → module	132	1	0
	42	bl	buffer reading → module	206	0.72	4
	43	request	buffer reading → HTTP	2,217	0.52	0
	44	ws	buffer reading → HTTP API	2,449	0.07	1
	45	floody	buffer reading → HTTP API	94	0.8	0
	46	tunnel-agent	buffer reading → HTTP API	225	1	0
Client-side issues	47	History sniffing [RJLS10]	HTMLElement.color → img.src	42	0	3
	48	Font fingerpr. [Aca+13]	HTMLElement.offsetWidth → img.src	145	0.5	1
	49	Font fingerpr.[3]	HTMLElement.offsetWidth → img.src	44	0.02	3
	50	Font fingerpr.[4]	HTMLElement.offsetWidth → img.src	134	1	0
	51	Browser ext. fingerpr. [SAS17]	HTMLElement.offsetWidth → request.open	1,451	1	1
	52	DoNotTrack leakage[5]	navigator_doNotTrack → HTMLElement.html	20	0	1
	53	Login state leakage[6]	onload event → document.innerHTML	191	1	0
	54	Engine fingerpr.[7]	HTMLElement.type → console.log	129	0	1
	55	Browser ext. fingerpr.[8]	onload event → HTMLElement.innerHTML	37	0	0
	56	Resource fingerpr.[9]	onload event → console.log	43	0	0

TABLE 8.1: Insecure programs, security policies, program size, sensitive branch coverage and number of upgrades. "module" stands for the module interface.

Our goal is an *in-depth* study of the different kinds of information flows for a range of security policies; we do not claim to study a representative sample of JavaScript applications. Existing *in-breadth* empirical studies, which analyze hundreds of thousands of web pages against fixed policies, provide clear evidence for security and privacy risks in JavaScript code [LSJ13; Mel+18; RJLS10]. In contrast to these large-scale studies, our effort consists in identifying vulnerable scripts from different domains and analyzing the flows therein.

INJECTION VULNERABILITIES Injection vulnerabilities are errors that enable an attacker to inject and execute malicious code. In Chapter 4 we demonstrate the devastating impact of injection vulnerabilities on server-side programs, e.g., when an attacker-controlled string reaches powerful APIs such as `exec` or `eval`. Such attacks can severely compromise integrity, e.g., deleting all files in a directory or completely controlling the attacked machine. We study 19 NODE.JS modules that contain injection vulnerabilities (IDs 1 to 19 in Table 8.1). As security policies, we consider the interface of a module as an untrusted source and the APIs that interpret strings as code, such as `exec` or `eval`, as trusted sinks.

REDOS VULNERABILITIES Regular expression Denial of Service, or ReDoS, is a form of algorithmic complexity attack that exploits the possibly long time of matching a regular expression against an attacker-crafted input. As discussed in Chapter 5, the single-threaded execution model of JavaScript makes JavaScript-based web servers particularly susceptible to ReDoS attacks. We analyze 19 web server applications that are subject to ReDoS attacks (IDs 20 to 39 in Table 8.1). As a security policy, we consider data received via module's interface as untrusted sources and regular expressions known to be vulnerable as trusted sinks.

BUFFER VULNERABILITIES Buffer vulnerabilities expose memory content filled with previously used data, e.g., cryptographic keys, source code, or system information. In NODE.JS, such vulnerabilities occur when using the `Buffer` constructor without explicit initialization. Buffer vulnerabili-

3 https://www.privacytool.org/AnonymityChecker/
4 http://www.lalit.org/lab/javascript-css-font-detect/
5 https://browserleaks.com/js/donottrack.js
6 https://robinlinus.github.io/socialmedia-leak/
7 https://www.privacytool.org/AnonymityChecker/
8 https://popmyads.com/
9 https://browserleaks.com/firefox#more

ties are similar to the infamous Heartbleed flaw in OpenSSL [Dur+14], as both allow an attacker to read more memory than intended. We analyze 7 applications subject to buffer vulnerabilities (IDs 40 to 46 in Table 8.1). The security policy requires that no information flows from the buffer allocation constructor to HTTP requests without initialization.

DEVICE FINGERPRINTING AND HISTORY SNIFFING Web-based fingerprinting collects device-specific information, e.g., installed fonts or browser extensions, to identify users [Aca+14]. History sniffing attacks use the fact that browsers display links differently depending on whether the target has been visited [RJLS10; Wei+11]. We analyze 10 client-side JavaScript applications that are subject to various forms of fingerprinting and history sniffing attacks (IDs 47 to 56 in Table 8.1). The security policies label as secret the sources that provide sensitive information, e.g., the font height and width, and as public sinks the APIs that enable external communication, e.g., image tags. We adapt these programs to our NODE.JS-based infrastructure by introducing minimal changes that emulate DOM interactions. We carefully cross-checked this adaptations in a pair-programming fashion, ensuring that all flows in the original program are preserved. The policies are application-specific and mark certain nodes in the emulated DOM as sources and sinks. In contrast to the other benchmarks, these programs can potentially be malicious [Nik+13; RJLS10]. That is, the assumption that the analyzed code is trusted does no longer hold.

8.3 METHODOLOGY

To enable our empirical study, we present a methodology that combines a set of novel metrics with a dynamic information flow analysis [HS12; Hed+14], a monitoring strategy [AF10], and an automated mechanism to insert upgrade statements [BHS12]. The metrics summarize the flows observed during the program execution. This section provides the necessary background on information flow analysis, an informal description of our methodology, and definitions of the metrics. It also presents a formalization of the core of our methodology.

8.3.1 *Setting: Information Flow Analysis*

SECURITY LABELS An information flow analysis associates each value with a *security label* that indicates how sensitive the value is. Labels are

Strategy	Sec. condition	Tracked flows			Permissiveness
		Expl.	Obs.	Hid.	
Taint tracking	Explicit secrecy	✓			Stop when H-labeled value reaches sink.
Observable tracking	Observable secrecy	✓	✓		Stop when H-labeled value reaches sink.
No Sensitive Upgrade	Non-interference	✓	✓	✓	Stop when L-labeled variable is written in sensitive context.
Permissive Upgrade	Non-interference	✓	✓	✓	Stop when partially leaked value is used.

TABLE 8.2: Monitoring strategies ("Expl." = explicit, "Obs." = observable implicit, "Hid." = hidden implicit).

typically arranged in a lattice [Den76]. To ease the presentation, we focus on two labels: H (for high or sensitive) and L (for low or insensitive), where H is more sensitive than L. Given a label $\ell \in \{H, L\}$, we write v^ℓ to denote that a value v has security label ℓ. If a value v does not have a label, we assume it is implicitly labeled as L.

INFORMATION FLOW POLICY The analysis checks whether data from a sensitive *source* influences data that arrives at an insensitive *sink*. The sources and sinks for a program are specified in an *information flow policy*, or short, *policy*. For integrity, the policy specifies that no information from untrusted sources (H) reaches trusted sinks (L). For confidentiality, the policy stipulates that no information from secret sources (H) reaches public sinks (L). We model sources by variables and object fields, and their security label corresponds to the label of the value that they contain initially. We denote sinks by a function `sink()`, which is implicitly labeled as L.

MONITORING STRATEGIES Different *monitoring strategies* for dynamic information flow analysis address the problem of checking whether an execution violates a policy. In this work, we focus on flow-sensitive dynamic monitors, where variables can be assigned different security labels during the execution. Table 8.2 gives an overview of the monitoring strategies studied in this chapter. Taint analysis tracks only explicit flows and stops the program only if an H-labeled value reaches a sink.

In contrast to taint tracking, the other three strategies also track implicit flows. The monitors identify implicit flows by maintaining a *security stack* that contains all sensitive labels of expressions in conditionals that influence the control flow. When the stack is non-empty, the program executes in a *sensitive context*. Observable Tracking [BSS17] tracks only explicit and observable implicit flows, but ignores hidden implicit flows. Whenever an L-labeled variable is updated in a sensitive context, observable tracking updates the label as sensitive and continues with the execution. For example, consider the following program, which is trivially secure because there is no call to `sink()`:

```
1 var location; var y; var z;
2 if (10 < location < 20) {
3   y = "Home"; }
4 //upgrade(y);
5 z = "You are at " + y;
```

Consider now an execution where the location is 15^H. Observable tracking updates the labels of `y` and `z` as sensitive and does not stop the execution.

The strictest monitoring strategies try to prevent also hidden implicit flows. We consider two variants of such a strategy. They both terminate the execution of the program whenever an observable implicit flow may lead to a hidden implicit flow in another execution. The No Sensitive Upgrade strategy (NSU) [AF09; Zda02] disallows updating the security labels of a variable in a sensitive context. In particular, it terminates the execution whenever such an update happens. For example, consider the execution of the above program with `location=`15^H. The NSU strategy terminates the program at line 3 due to the update of the L-labeled variable `y` in a sensitive context.

Permissive Upgrade (PU) [AF10] is a refinement of the NSU strategy. It labels a value as *partially leaked* if an L-labeled variable is updated in a sensitive context, and terminates the program if the updated variable is further used outside the sensitive context. Consider again the same execution of the above program. The PU strategy labels `y` as partially leaked at line 3 because the program writes to the L-labeled variable in a sensitive context, and then terminates the program at line 5 because the value is used. In our work, we use the PU strategy to study the prevalence of different kinds of flows.

UPGRADE STATEMENTS Naively applying the PU strategy to real-world programs can be very restrictive and risks to increase the number of false

positives, i.e., terminate many secure executions. To address this problem, Austin and Flanagan propose the *upgrade statement* [AF09] and the *privatization statement* [AF10]. These statements change the label of a variable to H explicitly, to signal a potential hidden implicit flow to the monitor. For example, we can insert an upgrade statement before line 5 in the above example to mark `y` as sensitive even if the branch is not taken. As a result, the program does not terminate immediately when the value is read. If the program would later call `sink(y)`, then the monitor would terminate the program and report a policy violation.

PERMISSIVENESS The above example illustrates the permissiveness issues of different monitoring strategies, i.e., that they terminate the program unnecessarily even though no policy violation occurs. Taint tracking and observable tracking both do not terminate the program. In contrast, both NSU and PU terminate the program unnecessarily. This overapproximation of policy violations is necessary to avoid potential hidden implicit flows. Adding upgrade statements avoids such premature termination of the program by assigning an H-label to `y`, independently of what branch of the conditional statement is executed. If we uncomment line 4, the execution proceeds without terminating the program unnecessarily. That is, upgrade statements may increase the permissiveness, but impose the cost of adding upgrade statements.

8.3.2 *Security Metrics*

Our approach uses program testing to measure the prevalence of different kinds of information flows. The basic idea is to test a program with an information flow monitor that implements the PU strategy, while incrementing counters that represent the number of explicit, observable implicit, and hidden implicit flows. These counters then allow us to reason about the prevalence of the different kinds of flows and about the policy violations that different monitoring strategies would detect. In contrast to the PU monitor that terminates the program when it encounters a policy violation, our monitor continues the execution to measure flows in the remainder of the execution. We refer to Section 8.3.3 for the formal definition of the monitor.

We consider information flows at two levels of granularity. On the one hand, we consider flows induced by a single operation in the program (Section 8.3.2.1). We call such flows *micro flows* or simply flows. Studying flows

at the micro flow level is worthwhile because it provides a detailed understanding of the operations that contribute to higher-level flows. In particular, flows provide a quantitative answer to the permissiveness challenges faced by state-of-the-art dynamic monitors that implement the NSU or the PU strategy. On the other hand, we consider transitive flows of information between a source and a sink, called *source-to-sink flows* (Section 8.3.2.3). Studying flows at this coarse-grained level is worthwhile because source-to-sink flows are what security analysts are interested in when using an information flow analysis.

The metrics presented in this section measure the prevalence of flows quantitatively, and do not attempt to judge the importance of flows. To ensure that our flows represent relevant problems, our study uses real-world security problems and policies that capture these issues.

8.3.2.1 *Micro Flows*

To measure how many explicit, observable implicit, and hidden implicit flows exist, our monitor increments the counters for these micro flows as follows.

EXPLICIT FLOWS The approach counts an explicit flow for every assignment event where the written value is sensitive but the value that gets overwritten (if any) is not sensitive. The rationale is to capture program behavior where sensitive information flows to a memory location that stores insensitive information. In contrast, overwriting a sensitive value with another (in)sensitive value does not leak any new information, and therefore does not count as an explicit flow.

For example, consider this code:

```
1 var x = 3^H; var y = 5^H; var z;
2 x = y; // no explicit flow
3 z = x; // explicit flow
```

OBSERVABLE IMPLICIT FLOWS The approach counts an observable implicit flow for every assignment event that happens in a sensitive context and that overwrites an insensitive value. Similar to explicit flows, the rationale is to capture program behavior that writes sensitive information to a memory location that stores insensitive information. The main difference is that the assignment happens because of a control flow decision made based on a sensitive context. Note that it is irrelevant whether the writ-

ten value is sensitive because the fact that a write happens leaks sensitive information.

For example, consider this code:

```
1 var x = true^H; var y = 3; var z;
2 if (x)
3   y = 5; // observable implicit flow
4 z = 7;   // no flow
```

HIDDEN IMPLICIT FLOWS The approach counts a hidden implicit flow for every execution of an upgrade statement of a variable containing insensitive information. The rationale is to capture assignment events that did not happen, but that could have happened during the execution if a control flow decision that depends on a sensitive value would have been different.

For example, consider this code:

```
1 var x = false^H; var y; var z;
2 if (x)
3   y = 5;         // not executed, no flow
4 upgrade(y);      // hidden implicit flow
5 z = y;           // hidden implicit flow
```

8.3.2.2 *Label Creep*

As mentioned earlier, a common reason for false positives is label creep. Since measuring false positives would be subject to a given source-to-sink policy, we focus on measuring the prevalence of the more general phenomenon of label creep in micro flows. Recall that this concept refers to the fact that information flow analysis may quickly label a large portion of all values handled in a program as sensitive. In most of the cases, this leads to an explosion in false positives that in turn reduces the usefulness of the analysis. We propose a novel metric called *Label Creep Ratio* (LCR) to assess how many variables and object fields in memory are labeled as sensitive.

$$LCR = \frac{\text{\# sensitive variables/fields ever assigned}}{\text{\# variables/fields ever assigned}}$$

For a given monitoring strategy, the Label Creep Ratio is the ratio between the number of assignments of H-labeled values and the total number of assignments. Intuitively, measuring the LCR throughout an execution es-

timates the speed at which the memory locations get assigned sensitive labels.

8.3.2.3 *Source-to-sink Flows*

To what degree do different kinds of flows contribute to policy violations? To address this question, we consider transitive flows from a source of sensitive information to a sink of insensitive information. For instance, none of the flows in the examples above correspond to a source-to-sink flow, since no sink statement is present.

Now, consider the code:

```
1 var x = false^H; var y; var z;
2 if (x)
3   y = 5;
4 upgrade(y); // hidden micro flow
5 z = x;      // explicit micro flow
6 sink(y);    // source-to-sink flow
```

The program contains two micro flows and one source-to-sink flow. However, if the execution is analyzed with taint tracking or observable tracking, the source-to-sink flow is missed, because it occurs only due to the upgrade statement.

As another example, consider the following code:

```
1 var x = true^H; var y; var z;
2 if (x)
3   y = 5;   // observable flow
4 z = x;     // explicit flow
5 sink(y+z); // source-to-sink flow
```

The source-to-sink flow will be detected by all three kinds of monitoring strategies, because the variable z gets labeled H via an explicit micro flow and then gets passed to the sink.

As illustrated by these two examples, we measure how many source-to-sink flows different monitoring strategies detect by tracking what micro flows contribute to a source-to-sink flow. Furthermore, to count the number of unique source-to-sink flows that a monitor detects, we compute the set of source code locations involved in each source-to-sink flow. If the code locations of two source-to-sink flows are the same, we count them as only one unique flow. This corresponds to the way a human security analyst would inspect warnings produced by an analysis.

8.3.2.4 *Inference of Upgrade Statements*

The approach described so far requires a program that indicates hidden implicit flows through upgrade statements. To obtain such a program, we adapt a testing-based technique for automatically inserting upgrade statements [BHS12]. The basic idea is to repeatedly execute the program with a particular policy, to monitor the execution for potentially missed hidden implicit flows (using the PU strategy [AF10], see Section 8.3.1), and to insert upgrade statements that signal them to the monitor when counting micro flows. Whenever the monitor terminates the program because it detects an access to a value u that is marked as partially leaked, the approach modifies the program by inserting an upgrade statement at the code location where u is next used; this upgrade statement in the modified program will then be executed whenever u is used again, regardless of whether the same branch that leads to the insertion of the upgrade statement is taken. The process continues until it reaches a fixed point, i.e., until the program has enough upgrade statements for the given tests.

The ability of our analysis to observe hidden implicit flows depends on the completeness of the inferred upgrade statements, since missing upgrade statements may result in false negatives for hidden implicit flows. How often this occurs depends on how well the analyzed executions cover the branches of the programs. One way to assess this ability would be to measure tradition branch coverage, i.e., the percentage of all branches that are covered by the given test inputs. However, traditional branch coverage is only of limited use because inserting upgrade statements does not rely on covering all branches in the code, but only on a subset. Specifically, the ability to insert upgrade statements depends on the branch coverage for conditionals that depend on sensitive values. We present a metric called *Sensitive Branch Coverage* (SBC) that captures this idea:

$$SBC = \frac{|\{c \in C \text{ where both true and false branch covered}\}|}{|C|}$$

where C is the set of conditionals that depend on a sensitive value. For example, consider executing the following program with `x=false`^{H}:

```
1 var x; var y
2 if (x)
3   y = 5;
```

The set C consists of the conditional at line 2, but since the execution covers only the false branch, $SBC = \frac{0}{1} = 0$.

8.3.3 *Formalization of Flows and Conditions*

We proceed by defining the syntax and semantics of NanoJS, a simplified core of JavaScript to illustrate the flow counting performed by our implementation.

Notation: We denote empty sequences by ε. Concatenating two sequences τ_1 and τ_2 is denoted by $\tau_1.\tau_2$. Slightly abusing notation, we also use the same notation to prepend a single element α to a sequence τ by writing $\alpha.\tau$. Similarly, we write $\alpha \in \tau$ to denote that α occurs in sequence τ.

NanoJS syntax: NanoJS statements:

$$\begin{aligned} \mathit{Stmt} ::= \;& \textbf{skip} \mid \varepsilon \mid c_1; c_2 \mid \textbf{sink}(e) \mid x = e \mid x[y] = e \mid \\ & \textbf{if } (e) \{ c_1 \} \textbf{ else } \{ c_2 \} \mid \textbf{while } e \textbf{ do } c \\ & \\ & \text{where } x, y \in \mathit{Name}, \text{ and } e \in \mathit{Expr} \end{aligned}$$

A terminated execution is denoted by ε. All function calls to sinks with expression e are modeled by $\textbf{sink}(e)$; other function calls are not considered in NanoJS.

Semantics: Operationally, the constructs in NanoJS behave as in standard imperative languages. To count micro flows, we associate each primitive value with a tuple $\kappa : \mathit{Cnt}$ of flow counts, where $\mathit{Cnt} = \mathbb{N}^3$. A tuple $(e, o, h) \in \mathit{Cnt}$ denotes e explicit flows, o observable flows, and h hidden flows. We write $\mathbf{0}$ for the tuple $(0, 0, 0)$. A value is either a primitive value annotated with a flow count, or an address on the heap. We assume that there is a set *Base* of primitive base types, such as boolean, numbers, and strings. A heap object $o \in \mathit{Obj}$ maps a finite set of names to values. We write **tt** for boolean value *true* and **ff** for boolean value *false*.

We use flow counts to track how information is propagated by a program, analogous to labels in other information flow monitors. We define a join-semilattice structure for flow counts as follows. Intuitively, a non-zero flow count indicates a sensitive value, whereas if all flow counts are zero, the value is insensitive: The join of two flow counts is defined as $\kappa_1 \sqcup \kappa_2 = \kappa_1 + \kappa_2$, where $\kappa_1 + \kappa_2$ denotes the pointwise addition of the two flow counts. We write $\bigsqcup t$ to denote the join operator over a list of flow counts t.

Two flow counts satisfy $\kappa_1 \sqsubseteq \kappa_2$ if whenever $\kappa_2 = (0,0,0)$ then $\kappa_1 = (0, 0, 0)$.

A configuration $\langle c, \rho, h, t, \kappa \rangle$ consists of a statement $c \in \mathit{Stmt}$, an environment $\rho : \mathit{Name} \to \mathit{Value}$ mapping variable names to values, a heap $h : \mathit{Addr} \to \mathit{Obj}$, a stack of security levels $t \in \mathcal{L}^\star$, and a sink counter $\kappa : \mathit{Cnt}$ counting flows reaching sink statements; we denote the set of configurations by Conf. An execution of a NanoJS program yields a trace $\mathit{Tr} = \mathit{Value}^\star$ indicating outputs produced by the execution.

We now define the small-step semantics of NanoJS. A step $\langle c, \rho, h, t, \kappa \rangle \xrightarrow{\tau} \langle c', \rho', h', t', \kappa' \rangle$ denotes a single evaluation step producing trace τ. We write ε for a terminated execution. Slightly abusing notation, we define $\bigsqcup(h, v)$ as the join of all labels occurring in value v with heap h. For simplicity, we assume that there are no cyclical references on the heap.

The function $\mathit{upgrade}(x, \rho, h)$ denotes the pair (ρ', h'), where the hidden flow count of all components of the value of a variable $x \in \mathit{Name}$ is incremented by 1. To update flow counts, we use an auxiliary function $\Delta : \mathit{Cnt} \times \mathit{Cnt} \times \mathit{Cnt}^\star \to (\mathit{Cnt})$. Intuitively, $\Delta(\kappa_{old}, \kappa_{new}, t)$ increments the explicit and observable flow counters for assigning a value with flow count κ_{new} to a location with label κ_{old} while the security stack is t. We define $\Delta(\kappa_{old}, \kappa_{new}, t) = (\Delta_e, \Delta_o, 0)$ where $\Delta_e = \begin{cases} 1 & \kappa_{new} \not\sqsubseteq \kappa_{old} \\ 0 & \text{otherwise} \end{cases}$ and $\Delta_o =$

$$\begin{cases} 1 & \kappa_{old} = \mathbf{0} \wedge \bigsqcup t \neq \mathbf{0} \\ 0 & \text{otherwise} \end{cases}$$

To define observations based on references passed to sinks, we use a helper function $\mathit{toVal}(h, v) : \{\mathit{Base} \times \mathit{Name}^\star\}$ that, given a value, returns all references to heap objects reachable from the value.

We denote evaluating an expression e in environment ρ and heap h by $[\![e]\!](\rho, h)$. The rules propagate flow counts into the result values; for example, adding two values with one explicit flow each will result in two explicit flows in the result. We assume, contrary to real-world JavaScript, that expressions do not have side effects.

Finally, Figure 8.2 gives the rules of small-step operational semantics for NanoJS with flow counting. The way the rules modify the environment and heap is standard. Some standard rules are omitted and provided in the appendix of the original publication [Sta+19]. In addition to the standard execution of a program, the semantics also track flow counts for each value. For example, an assignment statement $x = e$ propagates the flow

E-Assign

$$\frac{[\![x]\!](\rho,h) = v_x \qquad \kappa_x = \bigsqcup(h, v_x) \qquad [\![e]\!](\rho,h) = v^{\kappa_e} \qquad \kappa' = \kappa_e + \Delta(\ell_x, \ell_e, t) \qquad v' = v^{\kappa'} \qquad \rho' = \rho[x \mapsto v']}{\langle x = e, \rho, h, t, \kappa\rangle \to \langle \varepsilon, \rho', h, t, \kappa\rangle}$$

E-If

$$\frac{[\![e]\!](\rho,h) = v^{\kappa'} \qquad i = \begin{cases} 1 & v = \mathbf{tt} \\ 2 & \text{otherwise} \end{cases}}{\langle \mathbf{if}\ (e)\ \{\ c_1\ \}\ \mathbf{else}\ \{\ c_2\ \}, \rho, t, \kappa\rangle \to \langle c_i\ ;\ \mathbf{pop}, \rho, \kappa'.t, \kappa\rangle}$$

E-Sink

$$\frac{[\![e]\!](\rho,h) = v^{\kappa''} \qquad \kappa_a = \bigsqcup(v,h) \qquad \kappa' = \kappa + \kappa_a + \Delta(\mathbf{0}, \kappa_a, t)}{\langle \mathbf{sink}(e), \rho, h, t, \kappa\rangle \xrightarrow{toVal(h,v)} \langle \varepsilon, \rho, t, \kappa'\rangle}$$

E-UpgradeL

$$\frac{[\![x]\!](\rho,h) = v^{\mathbf{0}} \qquad v' = v^{(0,0,1)} \qquad (\rho', h') = upgrade(x, \rho[x \mapsto v'], h)}{\langle \mathbf{upgrade}(x), \rho, h, t, \kappa\rangle \to \langle \varepsilon, \rho', h', t, \kappa\rangle}$$

Figure 8.2: Rules for NanoJS with flow counting.

counts of the assigned expression e and additionally increments the explicit flow count if e has non-zero flows and the observable flow count if the control-flow path is determined by sensitive data. A sink statement $\mathbf{sink}(e)$ increments global counts representing source-to-sink flows. Since all sink statements model writes to *insensitive* sinks, any write of an expression with non-zero flow counts will result in incrementing the global counters.

Security conditions: We also adapt existing security conditions for tracking only explicit or observable flows to NanoJS [BSS17]. To capture only explicit flows, we use the notion of *explicit secrecy*; intuitively, a run of a program satisfies explicit secrecy if and only if the program obtained by sequentially composing all non-control-flow commands executed during that run does not leak information. For example, the program $\mathbf{if}\ (h)\ \{\ l = 1\ \}\ \mathbf{else}\ \{\ l = 2\ \}\ ;\ \mathbf{sink}(l)$ would produce the extracted programs $l = 1\ ;\ \mathbf{sink}(l)$ or $l = 2\ ;\ \mathbf{sink}(l)$ depending on the value of h in a given run. In both cases, the extracted program contains prohibited information flows, since the source program only leaks information through an *implicit* flow.

To track only explicit and observable implicit flows, we keep branching constructs in the extracted program, but replace not taken branches by **skip**. If the extracted program does not leak sensitive information, then

the run satisfies *observable secrecy*. For example, in the program $l = 0$; **if** (h) { $l = 1$ } **else** { **skip** } ; **sink**(l), observable secrecy would extract either $l = 0$; **if** (h) { $l = 1$ } **else** { **skip** } ; **sink**(l) or $l = 0$; **if** (h) { **skip** } **else** { **skip** } ; **sink**(l). This matches the intuition that an observable flow only occurs in the run where h is **tt**, where the assignment $l = 1$ is executed, but not in a run where h is **ff**, since this run only leaks information through a hidden implicit flow; i.e., the extracted program when $h =$ **tt** leaks information, but the extracted program for $h =$ **ff** does not.

Soundness: To establish soundness of our counting scheme, we show that if all explicit flow counts for all sinks for a given run are 0, then that run satisfies explicit secrecy. Similarly, we show that if all explicit and observable flow counts are 0, the run satisfies observable secrecy. The formal theorem statements and proofs can be found in appendices of the corresponding publication [Sta+19].

8.3.4 *Implementation*

To implement our methodology, we develop a tool for dynamic information flow analysis following Hedin et al. [HS12; Hed+14]. The implementation builds on Jalangi [Sen+13], a dynamic analysis framework for JavaScript that uses source-to-source transformation. Since Jalangi supports ECMAScript 5 only, we down-compile programs written in newer versions of the language with Babel [Bab]. Building on top of Jalangi allows us to focus on the important parts of the analysis and let the framework handle otherwise challenging aspects of implementing a dynamic information flow analysis, e.g., on the fly instrumentation of code produced by `eval`, exceptional termination of functions, boxing and unboxing of primitive values [CN15]. We handle higher-order functions and track dynamic modification of object properties as described by Hedin and Sabelfeld [HS12]. Our policy language is expressive, allowing the security analyst to mark both functions and arguments of callbacks as sources.

To approximate the effects of native calls, we model them by transferring the labels from all parameters to the return value. Moreover, if one of the parameters is an object, we propagate labels from all its properties to the return value. For a set of frequently used native functions, such as `Array.push`, `Object.call`, and `Object.defineProperty`, we create richer models that propagate labels more precisely. To increase the confidence in our implementation, we created more than 100 validation tests that assert the correctness of label propagation in typical usage scenarios.

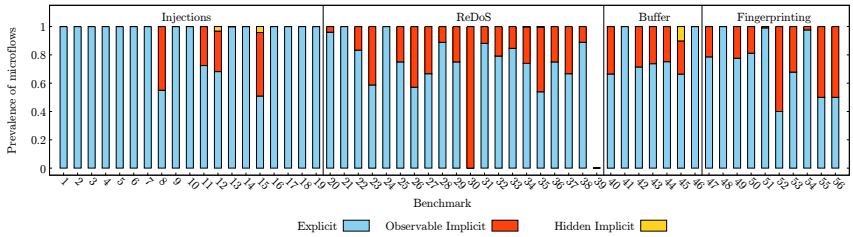


FIGURE 8.3: Prevalence of micro flows.

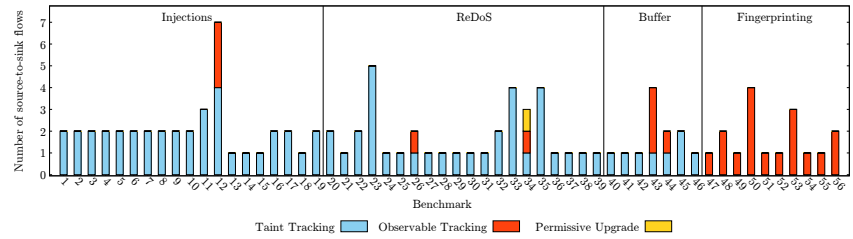


FIGURE 8.4: Number of source-to-sink flows detected at different security modes.

When inserting upgrades, the implementation does not modify the actual source code but it stores the source code locations of upgrades, and then performs the upgrades at runtime.

8.4 EMPIRICAL STUDY

This section presents the results of our empirical study that assesses the costs and benefits of tracking different kinds of flows.

The last two columns of Table 8.1 show the sensitive branch coverage (SBC) and the number of upgrades inserted while executing the benchmarks. Overall, the tests used for the study reach a high SBC, for 54% of the programs even 100%, enabling the analysis to insert upgrade statements. For each of the considered benchmarks, our tool can detect source-to-sink flows. This is hardly surprising, since we already know that the programs contain such flows, but it shows that our tool can handle complex, real-life JavaScript code.

8.4.1 *Prevalence of Micro Flows*

At first, we address the question of how prevalent explicit, observable implicit, and hidden implicit micro flows are among all operations that induce an information flow. Figure 8.3 shows the distribution of micro flows for our benchmarks. The majority of benchmarks contain both implicit and explicit micro flows. Benchmark 39 is a special case where reaching the sink is the first operation performed on the untrusted data, and hence the data flows directly from source to sink without producing any micro flow. The explicit flows are by far the most prevalent, appearing in all but one benchmarks. Five benchmarks also contain hidden implicit flows, but we can safely conclude that these cases are rare.

8.4.2 *Source-to-Sink Flows*

We now evaluate source-to-sink flows, which are the ultimate measure of success for an information flow analysis. Source-to-sink flows are what a security analysts ultimately cares about: how does information from a sensitive source reaches an insensitive sink. Information flow analysis has no way to show that such a flow is security-relevant, but it is the analyst's job to further inspect the flows and decide. In this section, however, we have a different goal and setup: we start with a set of known security problems that produce a source-to-sink flow and proceed by showing what type of analysis is needed to detect these problems.

Our tool can enforce different security conditions (cf. Section 8.3.3). For example, if we are interested only in explicit and observable implicit flows, we can run the tool in observable tracking mode and enforce observable secrecy. Figure 8.4 presents the number of source-to-sink flows detected by different monitoring strategies. All the integrity vulnerabilities can be detected by taint tracking only, and all the security violations in our data set can be detected through observable tracking. Moreover, all the Node.js vulnerabilities can be detected by the taint tracking only, independently of whether they are confidentiality or integrity vulnerabilities. We argue that this is because our Node.js programs are expected to be trusted. That is, a security issue may arise from a programming error, but not by malicious intention. This assumption does not hold, however, for the fingerprinting and history sniffing benchmarks, where only observable implicit flows contribute to the source-to-sink flows. A second explanation for why the implicit flows are prevalent in the browser environment is that there are

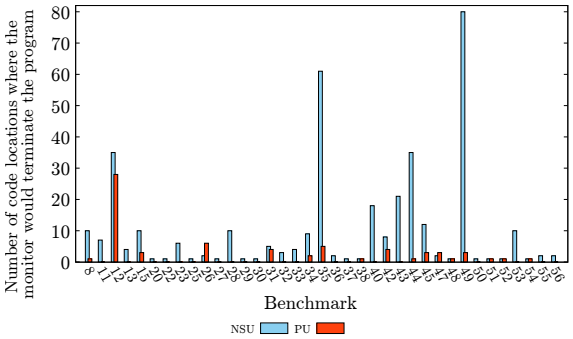


FIGURE 8.5: Number of violations, summarized by code locations, raised by NSU and PU monitors.

already a set of security mechanisms in the browser that prevent certain type of dangerous behavior. For example when fingerprinting the login state using images, an attacker cannot directly read the bytes of the image due to same origin policy, and hence it relies on measuring its width.

We analyzed in detail the additional source-to-sink flows detected by observable tracking for benchmarks 12, 26, 34, 43, and 44, and by PU for benchmark 34. In all these cases the reported flows are false positives, since they do not allow an attacker to exploit the respective vulnerability. In Section 8.4.4, we discuss in detail why these false positives occur when data is propagated through implicit flows.

Our results indicate that observable tracking is enough to tackle all the real-life security problems we consider and that taint tracking suffices for all the trusted code. We do not claim that there are no real-life security problems beyond observable secrecy, we just do not see any in our data set. Moreover, we believe that when strong controls are in place, attackers will be motivated to use more sophisticated attacks, possibly though the use of hidden implicit flows. However, tracking these flows is expensive as we will see in the remainder of this section.

8.4.3 *Permissiveness*

A potential problem for adopting information flow analysis in practice is its limited permissiveness, i.e., the fact that a monitor may terminate the program even though no data flows from a source to a sink. Our metrics allow us to quantify this effect both for the NSU and the PU monitoring

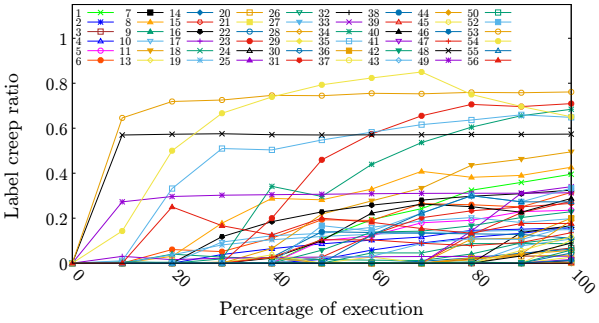
strategies. Specifically, we measure how many code locations a user would have to inspect because a monitor terminates the program. The NSU monitor terminates the program when an update of an insensitive variable is performed in a sensitive context. This condition corresponds to observable implicit micro flows and we count the number of code locations where such a flow occurs. The PU monitor terminates the program when an insensitive variable that was updated in a sensitive context is read. This termination condition corresponds to the locations where our tool inserts an upgrade statement. Figure 8.5 shows the number of code locations affected by the lack of permissiveness for NSU and PU. We exclude benchmarks for which neither of the monitoring strategies raises an alarm. On average, NSU throws 5.46 times more alarms than PU, that is, PU is much more practical than NSU. However, when comparing the PU violations to the number of source-to-sink flows that require PU (Figure 8.4), we observe that most of the PU alarms do not translate to actual source-to-sink flows and should be considered false positives.

8.4.4 *Label Creep Ratio*

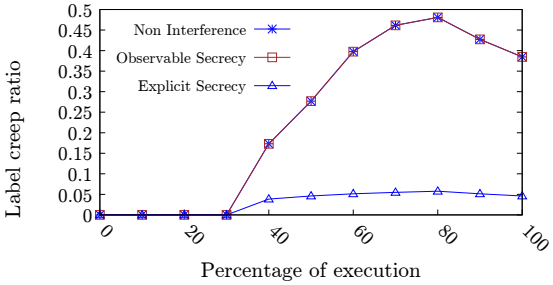
As a second metric for the cost of different kinds of flows, we use the Label Creep Ratio (LCR) defined in Section 8.3.2.1. For each benchmark and monitoring strategy, we measure how the LCR changes during the execution time. Figure 8.6a shows the ratio for PU monitoring. The metric is not monotonically increasing because the analysis is flow-sensitive, i.e., the security label of a variable may change over time. Nevertheless, the LCR steadily increases for most benchmarks, which confirms the label creep problem. Because our policies are targeted at detecting known security problems in the benchmarks, the maximum LCR reached is relatively low (20%, on average).

A comparison of different monitoring strategies shows that stricter monitoring causes more label creep. On average, observable tracking has a 0.3% smaller LCR than PU; a taint tracking analysis has a 45.4% smaller LCR than observable tracking. Figure 8.6b illustrates this effect with a representative benchmark (number 11). The graph shows how label creep increases for observable tracking compared to taint tracking.

We illustrate with the same benchmark 11 how label creep may translate to false positives. By revisiting Figure 8.4, we observe that the implicit flows do not contribute additional source-to-sink violations compared to a taint analysis. Figure 8.7 shows an excerpt of the source code of the



(a) Each benchmark



(b) Benchmark 11 at various security modes

FIGURE 8.6: Label Creep Ratio (LCR) over execution time

```
1  // query marked 'sensitive':
2  function parseQuery(query) {
3    // query pushed on the stack:
4    if(query instanceof Function) {
5      var nF = eval(query); // sink call
6      return [new Part(null, '$', nF)];
7    }
8  }
9  function Part(f, operator, operand, p){
10   if(p === undefined)
11     p = [];                          // implicit
12   this.field = f;                    // implicit
13   this.operator = operator;          // implicit
14 }
```

FIGURE 8.7: Implicit flows snippet from benchmark 11.

	Explicit Secrecy			Observable Secrecy			Non Interference		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Command injection	10	59,339	1,118,862	10	59,383	1,118,910	10	59,540	1,118,941
ReDoS vuln.	3	210	2,064	3	540	6,152	3	633	7,073
Buffer vuln.	98	5,740	24,690	98	6,007	24,748	98	6,084	24,843
Client-side progr.	4	5,919	40,364	14	19,555	134,765	16	20,890	136,502

TABLE 8.3: Number of instrumented operations handling sensitive data for different benchmarks and monitors.

benchmark. The code is vulnerable to code injection, where `query` is the source and `eval` is the sink. The source-to-sink flow is trivial since the sensitive data is directly passed to the sink at line 3, which a taint tracker easily detects. In addition, observable tracking pushes the query on the security stack at line 2, which causes implicit flows at lines 10 and 11 where two constants are written to memory. For detecting code injections, these flows are irrelevant. For example, suppose we have a statement `eval(this.operator)` at line 12, for which observable tracking would report a source-to-sink flow. This source-to-sink flow would be a false positive because the attacker can only control whether the call to `eval` happens, not what value flows into it.

8.4.5 *Runtime Overhead*

The last cost metric we use is a proxy measure for the runtime overhead imposed by different monitors. For each benchmark we count the number of operations that propagate a label or that modify the security stack. Table 8.3 shows how the number of events depends on the kind of monitor, aggregated by the different types of vulnerabilities we consider. As expected, raising the security bar translates into larger runtime overhead. Interestingly, this increase is not uniform across the different types of benchmarks. For injections, the cost increase is relatively small, while for ReDoS and client-side programs the increase between explicit and observable secrecy is more than 2.5-fold. We hypothesize that this is due to the structure of the programs: when comparing these findings with the micro flows in Figure 8.3, we see that implicit flows are more common in ReDoS and client-side programs than in injections. The price paid to track implicit flows in the client-side benchmarks translates to detected source-to-sink

flows, as we have seen in Section 8.4.2, while this is not the case for ReDoS vulnerabilities.

8.4.6 *Threats to Validity*

The validity of the conclusions drawn from our study is subject to several threats. First, our dynamic information flow analysis uses a simple model for native functions (Section 8.3.4), which may not accurately capture all effects of these functions. To minimize the influence of this limitation, we focus the study on subject programs that have relatively few native calls. We also wrote a set of precise models for some of the array and string native functions. Second, our results are limited to the programs we consider and may not generalize to other programs or classes of programs. In particular, we mostly consider non-malicious programs, where difficult-to-analyze flows may be less prevalent than in malicious code. Our methodology is generic enough to be easily applied to other programs. Finally, the hidden implicit flows that our methodology can observe are bounded by the upgrade statements inserted into the programs, which in turn depend on the tests we use to exercise the programs. To mitigate this threat we constructed tests in a way that increases the sensitive branch coverage. However, multiple paths cannot be covered due to a variety of reasons, e.g., error cases that cannot be easily triggered or unfeasible execution paths. Despite these limitations, our study produces interesting insights about the kinds of flows that appear in real-world JavaScript programs and the cost-benefit tradeoff of information flow analysis.

8.5 CONCLUSIONS

This chapter presents an empirical study of information flows in real-world programs. Based on novel metrics to capture the prevalence of explicit, observable implicit, and hidden implicit flows, as well as the costs they involve, we study 56 JavaScript programs that suffer from real-world security problems. Our results show that implicit flows are expensive to track in terms of permissiveness, label creep, and runtime overhead. We find taint tracking to be sufficient for most of the studied security problems, while for some privacy scenarios observable tracking is needed. Our work helps researchers, security analysts and analysis developers to better understand the cost-benefits tradeoffs of information flow analysis.

9

EXTRACTING SPECIFICATIONS FOR JAVASCRIPT LIBRARIES

In this chapter we present an automated technique for extracting taint specifications of JavaScript libraries. These specifications addresses the excessive code reuse problem (see particularity P_2 in the introduction) by better transmitting security assumptions from the library creator to its consumer, i.e., developers of client-side code reading the library documentation, or by enhancing the performance of existing static analysis tools. This chapter shares material with the corresponding publication [Sta+20].

9.1 MOTIVATION

An important characteristic of modern JavaScript-based applications is the extensive use of third-party libraries, as discussed in Chapter 2. On the npm platform more than 1 million packages (mostly libraries) are available,[1] and only a few of them have been screened intensively for security vulnerabilities. A challenge when analyzing the security of npm packages is that they are often not self-contained, but they in turn depend on other npm packages for providing lower-level functionality. In Chapter 2 we show that, on average, every npm package depends on 79 other packages and on code published by 39 maintainers. To correctly understand an application that uses npm packages, one needs to consider all these dependencies.

Two main directions are being pursued for automatically securing npm packages. First, there are tools that aggregate known security vulnerabilities in specific versions of individual libraries and report them to the developer directly. For example, `npm audit` analyzes all the dependencies of a Node.js application and warns the developer about any known vulnerability in the dependent-upon code. GitHub, Snyk, and other companies offer similar services, and related work [Lau+17] advertises such security controls. The main limitation of this approach is the high number of false positives. Often the critical part of the library is not used by the application, or it is used in a way that is completely harmless. For example, an applica-

1 http://www.modulecounts.com/

tion may use an npm module vulnerable to command injection attacks, but it only provides string constants provided by the developer as input to this module. We believe it is important to make the distinction between merely relying on a library that contains a potential known vulnerability and using that library in an insecure way. Another problem with these tools is that libraries that use insecure features of the JavaScript language or of the Node.js framework are often not registered as having "known vulnerabilities" if their documentation indicates that these features are being used internally. An example of such a library is the package `jsonfile` that provides functionality for easily accessing JSON files. Even though such a library is not considered vulnerable by itself, it may be used in an insecure manner, e.g., by propagating attacker-controlled data into file system paths.

A more precise approach for security of JavaScript applications pursued both by academia and by industry practitioners is static program analysis. In taint analysis, which is a kind of program analysis that can in principle detect most common forms of security issues, security properties are expressed as direct information flows from *sources* to *sinks*: either from untrusted sources to sensitive sinks (integrity) or conversely from sensitive sources to untrusted sinks (confidentiality). We focus on integrity because it covers the vast majority of security vulnerabilities reported by the community,[2] and we ignore indirect flows, also called implicit flows, because we show in Chapter 8 that they appear seldom in real-world npm vulnerabilities.

Modularity is the key to scalable static analysis. For example, the LGTM[3] analyzer by Semmle, a state-of-the-art taint analysis for JavaScript (and other languages), achieves high scalability by analyzing modularly. When analyzing one module of an application, other modules are either ignored or treated according to manually written specifications that describe essential taint flows where available. Ignoring modules leads to inaccurate analysis results, while manually constructing specifications is a demanding and error-prone task, so only a limited number of npm modules are considered. An important question hence is how to obtain specifications of modules in an automated way.

Inspired by Modelgen for Android [CAA15], we present a technique that dynamically infers explicit taint flow summaries for npm modules, to be utilized in a static analysis, such as LGTM. Besides being designed

2 https://www.npmjs.com/advisories
3 https://lgtm.com/

for JavaScript, our technique is more general than Modelgen, allowing for complex summaries to be extracted. For example, we are the first to support summaries involving callback arguments and instantiated exported classes. Moreover, our technique considers the large amount of transitive dependencies in npm and thus allows the extraction of summaries for multiple npm packages in the same execution.

Another source of inspiration is the NoRegrets tool [MMT18] that leverages the vast number of open source packages available in the npm repository to obtain information about how the most important libraries are being used. Many of those packages have test suites, and by running the test suite of a package we can gain information about the taint flows in all the packages it depends on, both directly and transitively.

A central technical challenge for adapting the Modelgen idea to our setting is that JavaScript is a highly dynamic language, which makes it non-trivial to map observations from dynamic analysis to taint specifications that fit into a static analysis. To this end, we adopt the notion of dynamic access paths from NoRegrets, allowing us to identify entry and exit points of taint flow in the libraries. Our dynamic analysis uses a variant of membranes [CM10; GMP14; KT15; Mil06] for tracking the taint flow between libraries and clients. It identifies flows between entry and exit points (propagations), between entry points and existing sinks (additional sinks) and between existing sources and exit points (additional sources). Finally, we propose deploying one membrane per npm module and hence extracting summaries for multiple modules at once.

We show that our approach is highly scalable by successfully running our dynamic analysis on 15,892 clients of 751 packages. The dynamic analysis is efficient, spending, on average, only 112 seconds per successfully analyzed client or 302 seconds per inferred specification. In total, it extracts 146 additional taint sinks and 7,840 propagation summaries spanning 1,393 modules. 35% of the summaries contain complex taint flows, such as between an argument of an exported method and a parameter passed by the library to a callback. The evaluation also shows that the extracted summaries can improve static analyses by enabling it to reveal otherwise missed vulnerabilities: 136 new alerts are produced, many of which correspond to likely vulnerabilities.

In summary, our contributions are:

- We present a novel, highly-scalable specification extraction technique for JavaScript libraries that builds on a dynamic taint analysis and leverages existing test suites.

```
let userInput = {
        tempDir: "./path/to/dir",
        cacheDir: "./path/to/cache"
}
const _ = require("lodash");
const rimraf = require("rimraf")

let obj = _.forIn(userInput, function(value) {
 rimraf(value, function(err) {
     if (err)
         console.log(err)
 })
})
```

FIGURE 9.1: A typical example of JavaScript code that uses npm modules. With dotted/blue we mark exit points from the client code and with solid/orange the entry points.

- We report our results from an extensive experimental evaluation of the approach. The results show that the dynamic analysis is able to infer non-trivial and accurate taint flow models in widely used NODE.JS modules.
- We demonstrate that the inferred taint specifications can be integrated into an existing static analysis tool, thereby enabling discovery of previously unknown security vulnerabilities.

Let us consider the example in Figure 9.1. This code fragment uses two of the most popular npm modules: `lodash`, a general-purpose utility library, and `rimraf`, a simple library for recursively deleting directories on the disk. In the presented example, the `forIn` method from `lodash` is used to iterate through the values of each property on the user input object. Each of these values are then passed to the `rimraf` module.

A human or an automated tool that aims at analyzing the code fragment in Figure 9.1 must first understand the essential semantics of the two modules. For example, one needs to understand that if some user input is passed to `rimraf` without sanitization, then it exposes a directory traversal vulnerability. However, this style of code can hinder understandability, both for unexperienced users and for static analysis tools. Specifically, it may not be clear that by invoking the `forIn` method with two parameters – an object to be traversed and a callback function – the second parameter will be invoked with the property values of the first parameter as arguments.

One way to address this problem is to analyze the library code together with the client code using a whole-program dataflow analyzer. However,

that approach suffers from serious scalability issues. For example, the implementation of the apparently trivial `forIn` method spans across 32 files. In Figure 9.2 we show a subset of the code that needs to be analyzed. Statically analyzing such a large amount of highly dynamic code is extremely expensive and tends to give prohibitively imprecise results [AM14].

When trying to analyze the source code of the `rimraf` module, one is faced with even greater challenges, as illustrated by Figure 9.3. To reveal the directory traversal problem discussed earlier, one needs to show that there is an unsanitized flow from the first parameter of the `rimraf` function to one of the file system access methods, e.g., `fs.rmdir`. However, as shown by the example, the call to this method is dispatched using dynamically attached methods on the `options` object. Once again, to the best of our knowledge, existing static analysis tools for JavaScript are unable to successfully analyze such highly-dynamic code at scale and with adequate precision to be practically useful.

Modular analysis, as exemplified by the LGTM analyzer by Semmle, addresses this challenge by analyzing each package in isolation. If a package depends on other packages, those are either ignored or modeled using manually-written specifications that capture the essential dataflows. Relying on simple generic specifications, e.g., saying that whenever a parameter is tainted then so is the return value, would be too imprecise for this example and lead to the static analysis missing important flows. Since creating useful specifications manually is difficult and not scalable, efficient automated alternatives are needed.

Our approach leverages the information in the npm repository about packages and their dependencies, together with the package source code available on GitHub. For this specific example, both `lodash` and `rimraf` have numerous open-source clients, many with test suites. By dynamically analyzing the executions of those test suites, we can automatically learn useful specifications.

9.2 Taint specifications for modules

The specifications we are interested in summarize the taint-relevant information for entry and exit points of JavaScript libraries. For example, one can specify that the information from entry point A may flow into exit point B or that values passed to an entry point eventually reach a potentially dangerous operation.

```
/* In the file forIn.js */
var baseFor = require('./_baseFor'),
    castFunction = require('./_castFunction'),
    keysIn = require('./keysIn');
function forIn(object, iteratee) {
  return object == null
    ? object
    : baseFor(object, castFunction(iteratee), keysIn);
}
module.exports = forIn;
/* In the file _baseFor.js */
var createBaseFor = require('./_createBaseFor');
var baseFor = createBaseFor();
module.exports = baseFor;
/* In the file _createBaseFor.js */
function createBaseFor(fromRight) {
  return function(object, iteratee, keysFunc) {
    var index = -1,
        iterable = Object(object),
        props = keysFunc(object),
        length = props.length;

    while (length--) {
      var key = props[fromRight ? length : ++index];
      if (iteratee(iterable[key], key, iterable) === false) {
        break;
      }
    }
    return object;
  };
}
module.exports = createBaseFor;
/* ... skipped the other transitive dependencies ... */
```

FIGURE 9.2: The implementation of `lodash`'s `forIn` method. For space reasons, only two of the 31 dependent files are shown.

```
var fs = require("fs")
function defaults (options) {
  var methods = [
    'unlink', 'chmod', 'stat', 'lstat', 'rmdir', 'readdir'
  ]
  methods.forEach(function(m) {
    options[m] = options[m] || fs[m]
    m = m + 'Sync'
    options[m] = options[m] || fs[m]
  })
}
function rmdir (p, options, originalEr, cb) {
  defaults(options)
  options.rmdir(p, function (er) {
    cb(er)
  });
}
module.exports = function rimraf(p, options, cb) {
  options.lstat(p, function (er, st) {
    return rmdir(p, options, er, cb)
  });
}
```

FIGURE 9.3: Simplified source code for the `rimraf` module.

The careful reader may have observed that there is a duality between the exit points of the client code, e.g., in Figure 9.1, and the entry points of the library, e.g., in Figure 9.2. For example, the `userInput` argument in line 8 corresponds to the `object` parameter in line 5. We will refer to both an entry point and its corresponding exit point by using the term *contact point*. We also introduce an access path mechanism to uniquely identify each contact point.

The specifications described in the remainder of this section can in principle be produced in multiple ways: either manually or by using a static or dynamic analysis. Section 9.3 presents an automatic inference process based on dynamic analysis.

9.2.1 *Specifying Contact Points*

Inspired by previous work to detect breaking changes in npm package updates [MMT18; MT19], we propose using an access path mechanism for specifying contact points. An access path, or short *ap*, can be described as an S-expression, which is read from the innermost expression outwards. Each type of symbol corresponds to an operation in the JavaScript language.

```
ap ::= (root <uri>)
     | (member <name> <ap>)
     | (parameter <i> <ap>)
     | (return <ap>)
     | (instance <ap>)
```

The innermost subexpression of a path always contains a `root` symbol, which holds an URI that refers to the module. For space reasons, we use package names instead of package URIs. For example, `(root dotenv)` refers to the module that is loaded when calling `require('dotenv')`. The other symbols are `member` to refer to properties of objects, `parameter` that refers to the *i*'th parameter of a function, `return` that refers to the return value of a call, and `instance` that refers to constructed values. For example, the path `(parameter 0 (member forIn (root lodash)))` represents both the exit point in line 8 of Figure 9.1 and the first entry point in line 5 of Figure 9.2. In the remainder of this section we show how access paths can express different kinds of taint specifications.

We assume that a collection of so-called known sources and sinks is provided. For example, values obtained from network communication via the

Node.js standard library are commonly treated as sources, and arguments to `exec` and `eval` are usually sinks.

We are interested in three types of specifications: *additional sinks* when we observe a flow from an entry point to a known sink, *additional sources* when there is flow from a known source to an exit point, and *propagation summaries* when there is a flow from an entry point to an exit point. We will now proceed to describe each of them in detail.

9.2.2 *Propagation Summaries*

The propagation summaries, or propagations for short, specify how taint may flow in and out of a library's functions. For example, a propagation summary can specify that if a tainted value enters the library as a specific argument to a function, then specific exit points of the library, e.g., properties on the return value, should also be considered tainted. Having such information available allows program analyses to reason about the potential taint flows without needing to reanalyze the source code of the library for every client.

The most simple form of flow is from an argument of a function to its return value, either because the argument is returned directly, or because the argument is used in the computation of the return value. Other more complicated forms of flow may also occur. For example, if an argument is written to some internal state of the library, and this state is then returned from another function, then we have a taint flow from the argument of one function to the return value of another function, which can also be captured as a propagation summary.

A propagation summary consists of two access paths: one that represents the point in the library API where the tainted value enters, and one that represents the point where the tainted value exits. Consider Example 1 where a function `f` has a parameter `x` and returns an object that has a property `p` with a value obtained from the `p` property of `x`.

Example 1

```
1 //module m
2 function f(x) {
3   return { p : x.a };
4 }
5 module.exports.f = f;
```

Taint Specification

```
(member a (parameter 0 (member f (root m))))
                          ↓
(member p (return (member f (root m))))
```

The interesting taint flow for this code is modeled by the taint specification shown next to the example, which indicates that taint flows from `x.a` to the `p` property of `f`'s return value.

This way of expressing taint flows is sometimes inconvenient. For example, a common JavaScript pattern is to iterate through all the properties of an object, which means that the accessed property names differ from client to client. An example of this reflective pattern is seen in Example 2. With the current notion of propagation summaries, we can only express flows involving specific properties, but in this case the relevant property names depend on the clients. For this purpose we introduce a wildcard notation for referring to every property of an object: `(member * <ap>)`. For example, one may refer to all the properties of the `obj` parameter in the program example with `(member * (parameter 0 (root sum))` as shown in the taint specification of Example 2.

Example 2

```
1 //module sum
2 function f(obj) {
3   let sum = 0;
4   for (prop in obj) {
5         sum += obj[prop
                  ];
6   }
7   return sum;
8 }
9 module.exports.f = f;
```

Taint Specification

```
(member * (parameter 0 (member f (root sum))))
                        ↓
(return (member f (root sum))
```

As mentioned earlier, callbacks are common contact points in npm modules. Our specifications refer to callbacks by treating a parameter as a function. The following specification summarizes the part of the `lodash` library presented in Figure 9.2, using a callback parameter exit point:

```
(member * (parameter 0 (member forIn (root lodash))))
                        ↓
(parameter 0 (parameter 1 (member forIn (root lodash))))
```

This propagation says that the value of every property of the object passed as the first argument of the `forIn` function may flow into the first parameter of the callback passed as the second argument.

A final propagation pattern worth discussing is one that involves contact points with return values. In Example 3, the `padder` module exports a single anonymous function in line 2. However, this function in turn creates an object with a `lpad` property pointing to an internal anonymous function. This case corresponds to the factory method design pattern from

object-oriented literature. After invoking the main exported method of the module, a client obtains a reference to the internal object declared in line 3, which in turn allows her to invoke the internal anonymous function from line 4. Thus, there are two exit points of the `padder` library in the presented example: one that returns an object with an `lpad` method, in line 7, and one corresponding to that method itself, in line 5. The latter depends on the former, because an object with the `lpad` method is only exposed to the client through the first exit point, which in turn creates more entry and exit points for the `lpad` method.

Example 3

```
1 //module padder
2 module.exports = function() {
3   let res = {};
4   res.lpad = function(s) {
5         return " " + s;
6   }
7   return res;
8 }
```

Taint Specification

```
(parameter 0 (member lpad (return
(root padder))))
        ↓
(return (member lpad (return (root
padder))))
```

9.2.3 *Additional Sinks and Sources*

If a value passed into a library reaches a known sink, we say that the entry point through which the value entered is an additional sink. Intuitively, passing the value to that contact point or to the sink itself has the same security implications for the client of the library, hence a program analysis can treat them the same way.

Revisiting the source code of the `rimraf` library in Figure 9.3, we can observe that the value passed as first argument to the main library function ends up in `fs.rmdir()`, which is a known sink for directory traversal vulnerabilities. This method allows recursively removing any folder on the disk, hence if an attacker can control the value passed into it, she can cause serious harm on the system. Therefore, it makes sense to specify the contact point `(parameter 0 (root rimraf))` as an additional sink.

Conversely, if inside the library a tainted value is created which then escapes into the client code through an exit point, we say that the exit point is an additional source. Example 4 shows a simple module that performs a TCP request and invokes a callback whenever data is received from the target server. This data should be considered tainted since it comes from untrusted third-party computers, so it is reasonable to specify the

contact point `(parameter 0 (parameter 2 (root my-tcp))` as an additional source.

Example 4

```
1 //module my-tcp
2 module.exports = function (host, port, cb) {
3   const net = require('net');
4   const client = new net.Socket();
5   client.connect(port, host, function() {});
6   client.on('data', function(data) {
7     cb(data);
8   });
9 }
```

Even though our dynamic analysis presented in Section 9.3 can in theory extract all the three types of specification presented so far, our prototype implementation introduced in Section 9.5 only supports the extraction of propagations and additional sinks. The main reason for omitting extraction of additional sources is that existing security vulnerability reports for npm packages often involve additional sinks, for example CVE-2017-1000219 or CVE-2018-3772, but vulnerabilities caused by additional sources are less common.

9.3 Inferring Taint Specifications via Dynamic Analysis

We now present a technique for dynamically inferring taint specifications, i.e., propagation summaries and additional sinks, of the form described in Section 9.2. The goal is to find relations between entry points and exit points, between entry points and existing sinks, and between existing sources and exit points.

Figure 9.4 illustrates how our technique works for a single npm module. The arrows represent information flow, possibly spanning multiple methods and modules. When the test suites are executed, values are intercepted at entry points and tainted with a unique identifier per entry point. The taint inside the module is then propagated using a dynamic taint analysis. Whenever a tainted value reaches a sink or an exit point, an additional sink or a propagation summary, respectively, is generated. Similarly, if a value that is tainted by an internal source is observed at an exit point, an additional source is generated for that exit point. All taints are removed at exit points, so we only infer specifications for the library code and not for the client code.

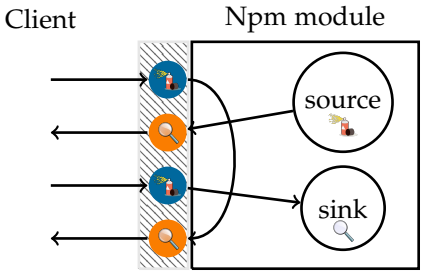


FIGURE 9.4: Inferring specifications for a single library: we taint the entry points and the sources and check taint at sinks and at exit points. The arrows show information flows and the shaded gray area represents the membrane.

Previous work [CAA15] considers arguments of methods in the public API as entry points and return values as exit points, but as the motivating example shows, this is insufficient for many npm modules. JavaScript libraries interact with their clients in complex ways, e.g., through callbacks like the ones in Figure 9.1 or by allowing plugins to be configured inside the library. Therefore, it is non-trivial to determine where the library code starts and where the client code ends. One way to refer to this point of contact between components and thus to generalize the idea of entry and exit points is by using the concept of *membranes* [CM10; Mil06].

9.3.1 *Membrane-Based Analysis*

The main idea of a membrane is to interpose on every interaction between the client and the library. Moreover, every reference that passes through the membrane becomes part of it. Existing work describes how to implement membranes and how to use them for implementing generic policies such as "the library should never use the native module `fs`". However, to be useful in our setting, we need a way to distinguish between entry and exit points of the library and to uniquely refer to every such point in the membrane.

To rigorously define membranes, we first introduce a way of intercepting operations on a given value. To this end, we rely on proxies, a concept introduced in ECMAScript 6. A proxy $P(v)$ for a value v is a wrapper object that attaches traps to the wrapped value. Every operation applied to the proxy results in an invocation on the corresponding trap. For exam-

ple, property reads, property writes, standard operations, and constructor applications all result in their corresponding traps firing. The traps can modify the behavior of the operation or just perform observing operations such as logging.

A proxy can therefore observe operations applied to the wrapped value, and even decide to modify these operations. Our analysis uses proxies to perform taint-relevant operations before and after the proxied operation is executed. We also need a way to associate a unique address, i.e., an access path, to each proxy and to specify whether the proxy corresponds to an exit or an entry point:

Definition 1 *A contact point, denoted $\langle v, ap, d \rangle$, is a tuple consisting of a proxy $P(v)$ around a value v, an access path ap that uniquely identifies the contact point, and a direction bit d that specifies whether the contact point is an entry or an exit point.*

For simplicity, we abuse the notation for a contact point $\langle v, ap, d \rangle$ by using $\langle v \rangle$ whenever the access path and the direction are not relevant for the description. One can specify entry and exit points for a library by introducing proxies around exported API methods in the library source code. The challenge lies in automatically identifying *all* the values that need to be proxied for intercepting all the interactions between two npm modules. Membranes provide an elegant solution to this problem:

Definition 2 *A membrane $\mathcal{M}$ is a set of contact points interposed between a library ℓ and its clients. $\mathcal{M}$ is initialized with $\{\langle v_\ell, \texttt{(root}\ \ell\texttt{)}, \text{ENTRY} \rangle\}$, i.e., the contact point that wraps the main value v_ℓ exported by the library. For every value v that is passed into or returned by an existing contact point in $\mathcal{M}$, a new contact point $\langle v, ap_v, d' \rangle$ is added to the membrane, i.e., $\mathcal{M} := \mathcal{M} \cup \{\langle v, ap_v, d' \rangle\}$.*

The new access point ap_v is derived from the existing ap by picking the grammar rule from Section 9.2.1 that corresponds to the JavaScript operation v passed through at the exit point, e.g., a property access or a parameter to a function call. Similarly, the direction of the new contact point d' is derived from the direction of the original contact point d by using the following intuition: the direction changes for all the values that are passed as arguments to a method in the membrane. Let us consider a function object that is passed into an entry point of a library as an argument. Once it reaches the other side of the membrane, i.e., in the library code, it should be considered as an exit point for the library. In Table 9.1 we summarize all the possible operations on a contact point and the way to derive the

Operation	Existing contact point		New contact point(s)		Pre action	Post action
	Access path	Direction	Access path	Direction		
`require("foo");`	-	-	`ap = (root foo)`	ENTRY	-	-
`x.prop`	ap_x	ENTRY	`ap = (member prop` ap_x`)`	ENTRY	-	-
	ap_x	EXIT	`ap = (member prop` ap_x`)`	EXIT	-	taint(x.prop, ap)
`res = x(arg)`	ap_x	ENTRY	ap_{par}`=(parameter <i>` ap_x`)`	EXIT	taint(arg, ap_{par})	checkTaint(res, ap_x)
			ap_{ret}`=(return` ap_x`)`	ENTRY		untaint(res, ap_x)
	ap_x	EXIT	ap_{par}`=(parameter <i>` ap_x`)`	ENTRY	checkTaint(arg, ap_x)	taint(res, ap_{ret})
			ap_{ret}`=(return` ap_x`)`	EXIT	untaint(arg, ap_x)	
`res = new x(arg)`	ap_x	ENTRY	ap_{par}`=(parameter <i>` ap_x`)`	EXIT	taint(arg, ap_{par})	checkTaint(res, ap_x)
			ap_{ret}`=(instance` ap_x`)`	ENTRY		untaint(res, ap_x)
	ap_x	EXIT	ap_{par}`=(parameter <i>` ap_x`)`	ENTRY	checkTaint(arg, ap_x)	taint(res, ap_{ret})
			ap_{ret}`=(instance` ap_x`)`	EXIT	untaint(arg, ap_x)	

TABLE 9.1: Creation of contact points inside the membrane and the corresponding taint operations executed before and after the proxied operation. The direction indicates whether the proxy corresponds to an entry or an exit point. The taint(v, ap) action associates a taint corresponding to the access path ap to runtime value v. The checkTaint(v, ap) action recursively searches for tainted values in v, where the taint has the same root package as the access path ap. Finally, untaint(v, ap) recursively declassifies all the values in v that have a taint with the same root as the access path ap.

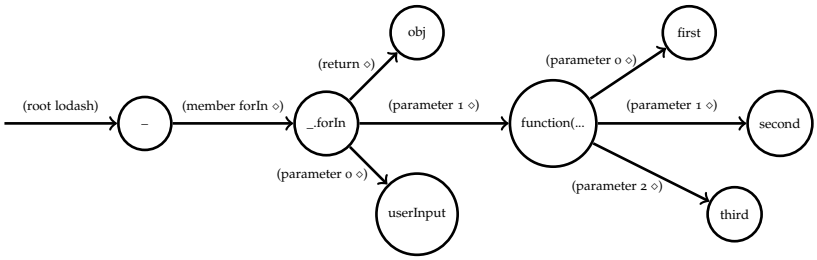


FIGURE 9.5: Contact points in the membrane between `lodash` and the client code in Figure 9.1.

access paths and direction bits for the new contact points. We also show the auxiliary operations necessary for tracking tainted values in the pre and post action columns. Note that both arguments and return values can be entry or exit points, depending on the direction bit.

To illustrate how contact points are created, consider the membrane between `lodash` and its client in Figure 9.1. The first contact point of the membrane is created when the library is required in line 5, i.e., $\mathcal{M} := \{\langle \_ \rangle\}$. When the `forIn` property is accessed in line 8 a new contact point is added to the membrane, $\mathcal{M} := \mathcal{M} \cup \{\langle \_.\texttt{forIn} \rangle\}$. When the accessed property is invoked in the same line, three contact points are created, i.e., $\mathcal{M} := \mathcal{M} \cup \{\langle \texttt{userInput} \rangle, \langle \texttt{function...} \rangle, \langle \texttt{obj} \rangle\}$. Finally, when the callback is invoked, three more contact points are created, one for each parameter. The access paths for each of these contact points are shown in Figure 9.5; they correspond to a derivation tree of the grammar in Section 9.2.1. To obtain the access path of a given contact point, one should traverse the tree from the root and replace all the ◇ symbols with the access path of the parent node. For example, the access path of $\langle \texttt{first} \rangle$ is:

```
(parameter 0 (parameter 1 (member forIn (root lodash))))
```

The dynamic taint analysis we use for propagating taint inside analyzed modules is fairly standard, with few idiosyncrasies. As noted earlier, we implement the taint-relevant operations described in the last column of Table 9.1 inside each module's membrane. These operations are in fact additional sources and sinks from the taint analysis' perspective since they either attach taint or check/remove taint. Once a property `p` is accessed on a value having a taint `t`, instead of directly propagating the taint, we create a new tainted value `(member p t)`. If the property `p` itself is also

tainted then we propagate the taint `(member * t)`. The intuition is that the tainted property comes from outside the module or from iterating through a tainted object, hence it should be considered as a generic access.

9.3.2 *Multi-Module Analysis*

Since an npm module can in turn use other npm modules, we propose deploying a membrane around each module to maximize the number of extracted specifications. We present this setup in Figure 9.6 in which module M interacts with two other modules: a direct dependency L and a plugin P. For now let us consider the relation between module M and the library L. Every entry point attaches a taint that uniquely identifies that entry point to each value that passes through it, e.g., entry point M_1 sets taint m_1. When a value passes through an exit point of a module, the analysis removes all the taints corresponding to that particular module. As a result, tainted values for module M can only live inside M or inside M's transitive dependencies, such as L. This behavior can be observed when following the information flow between entry point M_1 and exit point M_3. The taint m_1 is carried by the value all the way through module L until the exit point M_3. Our analysis infers two propagation specifications: $M_1 \rightarrow M_3$ and $L_1 \rightarrow L_3$.

Similarly, the value that enters through M_2 inside module M gets attached the taint m_2, and further enters through L_2 inside module L, where it gets attached taint l_2. Thus, when the value finally reaches the sink inside module L, it has two taints, m_2 and l_2. The analysis generates two additional sinks, for M_2 and for L_2, since from the client's perspective a value may flow from one of these entry points into an existing sink.

9.3.3 *Handling Plugins*

One may wonder whether or not a module's dependencies should be considered as part of the module's code as described in the previous section. We propose distinguishing between two types of dependencies: direct dependencies and plugins. A direct dependency is one that is required verbatim by the developer in the source code of the module, and a plugin is a dependency that is injected by the client code. For registering a plugin, a client needs to pass a reference to the plugin through the membrane. An example of this pattern can be observed in Example 5 that shows the popular `express` framework instantiated with the plugin `body-parser`. The

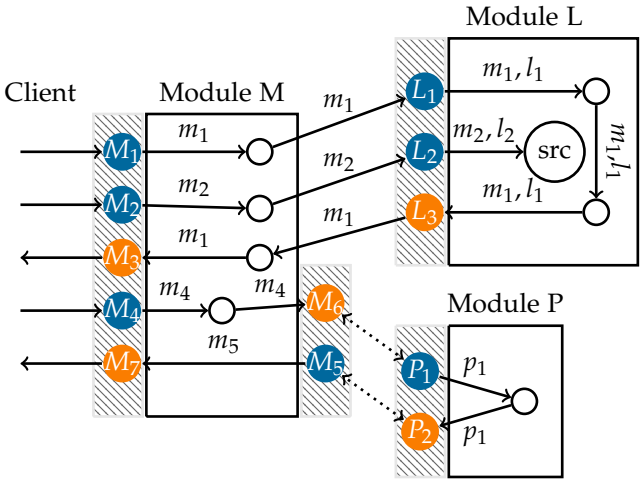


FIGURE 9.6: Inferring specifications for multiple modules at once: every entry point adds a unique taint and every corresponding exit point declassifies it. The semantics of shapes and colors are the same as in Figure 9.4. The dotted arrows depict equivalences between contact points.

client code loads the `body-parser` plugin and passes it to the `express` module through the `use` method that is part of `express`'s membrane.

Example 5

```
1 const express = require('express');
2 const bodyParser = require('body-parser');
3 const app = express();
4 app.use(bodyParser.json())
```

Treating plugins differently when extracting specifications is extremely important because we do not want to infer specifications that only apply when a certain plugin is loaded. Instead, we want the specifications to be as widely applicable as possible. Therefore, direct dependencies that are loaded inside the module are considered part of the code base of the module, while plugins are not.

Consider the relation between module M and its plugin P in Figure 9.6. When the value carrying the taint m_4 reaches the membrane that separates M from P the taint is removed; we say that the value is declassified. Overall, for the flow between M_4 and M_7 that passes through module P, our analysis infers three specifications: $M_4 \rightarrow M_6$, $M_5 \rightarrow M_7$, and $P_1 \rightarrow P_2$.

The main use case of the extracted taint specifications is for improving existing program analyses. Most importantly, taint specifications can be consumed by static analyses. The benefits of hybrid analyses, i.e., static plus dynamic, are thoroughly explored in the literature. Typically, a static analysis uses the results from a dynamic analysis, either to get a more precise result or to get coverage of code that is otherwise difficult to analyze statically. As mentioned in the introduction, industrial static analyses sometimes do not even try to analyze NODE.JS modules, but instead rely on manually written taint specifications or coarse-grained assumptions about taint flow in modules. However, such specifications are both error-prone and hard to maintain. In contrast, our specification generation analysis is fully automatic, and can therefore easily be re-run whenever modules are updated. Moreover, we show that there is a significant overlap between the specifications our analysis generates and existing manually-written models used by the commercial LGTM taint analysis, demonstrating that our analysis can infer precise module specifications that resemble and improve upon hand-written specifications.

Another use case for the extracted specifications is to serve as a form of documentation for the module they were extracted from. Effectively, they can act as a contract between module developers and module users that specifies, for example, who is responsible for sanitizing end user input. We observe that many security vulnerabilities reported by the community or by researchers, e.g., our vulnerabilities in Chapter 4, are actually additional sinks. For example, a typical vulnerability occurs when a user-supplied value is involved in constructing some string that is then executed by the `eval` function. In some unfortunate situations attackers can compose the user-supplied value in ways that enables executing malicious code. To warn users of modules about potential vulnerabilities, inferred specifications could be shown to developers. For example, an additional sink could inform the client that an argument passed to a specific method should be sanitized to prevent malicious code injection attacks.

Finally, we propose using the generated taint specifications for regression analysis. When a previously unobserved taint specification is suddenly generated for a new version of a library, e.g., a new additional sink appears, both the developer of the library and its clients should be alerted. Essentially, a change in a taint specification should be treated as a change

to the API. Automatically inferred specifications could help automate this kind of regression analysis.

9.5 EVALUATION

IMPLEMENTATION We implement the described specification extraction technique in a tool called TASER[4]. The dynamic analysis component is built on top of NodeProf [Sun+18], an instrumentation framework for NODE.JS. As a starting point for finding additional sinks, we mark 40 methods of the built-in JavaScript APIs as sinks. These methods cover five well-known security issues: command injection, code injection, directory traversal, regular expression injection, and NoSQL injection. We implement limited support for sanitizers by declassifying any information flow that passes through a function and an npm module whose name or dynamic access path contains specific strings, e.g., "escape" or "sanitize".

BENCHMARKS We apply TASER to 751 npm packages, all from the top-1000 most depended upon packages. Because some packages contain multiple modules and because TASER performs a multi-module analysis we analyze a total of 2,300 modules. For each analyzed npm package, we consider the 200 best rated clients, according to npm stars, and execute their test suites to analyze the execution with TASER. We stop a test suite after a timeout of 10 minutes. If available, we also use the test suite of the npm package itself for driving the dynamic analysis. Ignoring some clients that we currently cannot analyze, e.g., due to test frameworks TASER does not support, or due to limitations of our implementation, the evaluation covers 15,892 clients, out of which 5,707 clients trigger at least one creation of a tainted value.

RESEARCH QUESTIONS Our evaluation aims to answer the following research questions:

RQ1 How many taint specifications does TASER extract?

RQ2 How efficient is the analysis?

RQ3 Are the extracted specifications useful for statically analyzing the security of npm modules?

4 It is an abbreviation of the longer *TAint Spec ExtractoR*.

RQ4 How do the extracted specifications compare to manually created models of npm modules?

RQ5 How does a static analysis based on Taser-inferred specifications compare to the state-of-the-practice `npm audit` approach?

RQ1: Extracted Taint Specifications

For the 2,300 analyzed modules, Taser extracts 7,840 propagation summaries and 146 additional sinks. For 457 packages, the tool extracts at least one propagation summary, and for 118 packages, it extracts at least one additional sink. The overall amount of specifications shows that manually writing taint specifications for thousands of packages is highly impractical. Instead, Taser enables extracting specifications automatically and updating them regularly with little effort.

We also check whether the specifications extracted by Taser contain advanced language constructs not supported by previous work [AB16; CAA15]. To that end, we count every propagation summary that involves (i) instantiated objects, i.e., an `instance` symbol in one of its access paths, (ii) callbacks, i.e., two or more `parameter` symbols in one of its access paths, or (iii) nested API calls, i.e., two or more `return` symbols in one of its access paths. We find 595 propagation summaries with instantiated objects, 1,467 with callbacks and 1,578 with nested API calls. In total, at least 2,838 specifications, i.e., 35% of the total, could not have been extracted by those previous approaches (even if re-implemented for JavaScript).

RQ2: Efficiency of the Dynamic Analysis

Generating specifiactions is not something that should be done often, so having a relatively large one-time cost is acceptable in practice. However, over time new libraries are created and existing libraries are updated, so new taint specifications naturally have to be generated for those libraries. Therefore it is interesting to consider the computational cost of generating taint specifications. On average, it takes 112 seconds to run the test suite of one client with the dynamic analysis enabled. This number depends on many factors, such as the size of the test suite, and how many JavaScript statements are being executed. Thereby, regenerating specifications for updated libraries and generating specifications for new libraries can be done in only a few hours per library, which we consider acceptable

Rule ID	New alerts
js/command-line-injection	2
js/file-access-to-http	64
js/path-injection	29
js/reflected-xss	5
js/regex-injection	13
js/remote-property-injection	20
js/user-controlled-bypass	2
js/xss	1
Total	136

FIGURE 9.7: Improvements to LGTM's standard analysis; rule IDs are hyperlinked to their documentation.

for specifications that can be reused repeatedly by a static analysis and other applications.

RQ3: Usefulness for Static Analysis

We evaluate the usefulness of TASER-extracted taint specifications by integrating them into LGTM, a state-of-the-art, industrial static analysis platform. A free instance hosted at https://lgtm.com continuously checks more than 130,000 open-source projects (including thousands of npm modules) for security problems. LGTM reasons about third-party npm modules based on a limited number of manually created taint propagation models. We add the extracted specifications into the static analysis and measure how many additional security alerts the analysis reports.

Figure 9.7 shows the improvements gained from enhancing LGTM's standard security analysis suite with the additional sinks and propagation summaries extracted by TASER. The first column lists the LGTM rule ID; for instance, js/path-injection flags potential directory-traversal vulnerabilities. The second column shows the number of new alerts found by incorporating our additional sinks and propagation summaries. In total, TASER enables LGTM to find 136 otherwise missed potential security problems.

```
var rimraf = require('rimraf');
/* omitted */
function onDeleteFile(req, res) {
    var uuid = req.params.uuid,
        dirToDelete = uploadedFilesPath + uuid;
    rimraf(dirToDelete, function(error) {
        /* omitted */
    });
}
```

FIGURE 9.8: Example of a new alert found based on a TASER-inferred specification.

To better understand the quality of the added alerts, we randomly sample 30 of the new alerts (five for rules with five or more results, and all results for the other rules). We find that 24 of them were true positives in the sense that they exhibit flow from a source to a sink.[5] Of the six false positives, five were due to imprecision of the static analysis (and hence unrelated to TASER), and one was due to a spurious additional sink extracted by TASER.

Figure 9.8 shows a simple example of a newly identified alert for the `js/path-injection` rule, which originates from the `FineUploader/server-examples` project from GitHub. The `req` argument contains an HTTP request object, so the LGTM security analysis considers `req.params.uuid` to be untrusted data since it might originate from a malicious attacker. After being concatenated with another string, it is passed to the `rimraf` function, which (recursively) deletes the file system path denoted by this string if it exists. The value of `req.params.uuid` is not checked, so in particular it could contain "`..`" components, allowing an attacker to delete arbitrary files on the file system.

Even though the flow from source to sink is very simple, LGTM does not flag this out-of-the-box, since it does not have a model of the `rimraf` package, and its implementation is too complicated for the static analysis to model as explained above. Our additional sinks, however, identify the first parameter of `rimraf` as a taint sink for `js/path-injection`, allowing LGTM to flag this code.

As an example of the use of propagation summaries, we notice that four of the five new alerts for `js/remote-property-injection` we examined made use of the propagation summaries for `_.forEach`, a `lodash` function similar in style to `_.forIn`. These propagation summaries describe flow

5 How many of these new results correspond to exploitable security vulnerabilities is a different question, which we do not consider here.

through a callback parameter, underscoring the importance of supporting such summaries.

RQ4: Comparison with Manually Created Specifications

The standard LGTM security analysis suite already includes manually written models of many popular npm packages, including sinks and taint propagation rules. By examining our automatically extracted taint specifications for overlap with these manually written models, we find that 12 of our additional sinks and 40 of our propagation summaries correspond to existing models. On the one hand, this confirms that the specifications we extract are practically relevant. On the other hand, it also shows that the vast majority of the Taser-extracted specifications are not yet covered by manual models.

As one example, our dynamic analysis correctly identifies the first parameter of the single function exported by the `cross-spawn` package as a sink for `js/command-line-injection`. LGTM includes a manual model for this. Additionally, Taser identifies an analogous sink for the `win-spawn` package, a by now deprecated predecessor of `cross-spawn`. LGTM does *not* include a model for this, presumably because `win-spawn` is less popular than `cross-spawn`, and the LGTM analysis authors focused on popular packages in writing their models. Our automated approach is not limited by such considerations and can hence provide a much broader coverage.

RQ5: Comparison with Coarse-Grained Warnings

The current security practice in the npm community is to warn users whenever they are relying on a module with a known vulnerability, as implemented, e.g., in the `npm audit` tool. This approach suffers from two limitations. First, it is limited to previously known and reported vulnerabilities. Second, it often causes spurious warnings, as a warning is issued for every module that depends on a vulnerable module, independently of whether the first module's use of the second module is affected by the vulnerability.

We show that our approach can help address both these limitations. First, one can use Taser to automatically find vulnerabilities, i.e., unsanitized, undocumented additional sinks. To evaluate the effectiveness of this approach, we run Taser using benign inputs for 24 vulnerable packages used in Chapter 7. Our approach finds additional sinks in 11 of the 24 packages. Limitations of the existing policy, i.e., missing sources, and in-

```
var printer = require("printer");
var benignInput = "printerName";
printer.printDirect({
    data: "Test",
    printer: benignInput,
    success: function (jobID) {
        console.log("sent to printer with ID: " + jobID);
    },
    error: function (err) {
        console.log(err);
    }
});
```

FIGURE 9.9: Benign input for the vulnerability described in npm advisory number 27.

sufficient modeling for arrays are the reasons why we do not find the rest of the sinks.

Second, TASER-extracted specifications can help identify the problematic entry point of a vulnerable library. This can reduce the false positive rate of the `npm audit` solution by only reporting an alarm when user-controlled values can reach that entry point. While implementing a more precise replacement for `npm audit` based on TASER-extracted specifications is out of scope for this work, we illustrate its potential effectiveness with the vulnerability in Figure 9.9. The example shows benign inputs passed to a module that suffers from a known vulnerability.[6] TASER infers the following additional sink for the vulnerable module:

```
(member printer (parameter 0 (member printDirect (root
                    printer))))
```

Similarly to SYNODE, proposed in Chapter 7, instead of alerting all users of the `printer` module, as `npm audit` currently does, the extracted specification could help raise an alarm only for users that call the vulnerable entry point with a non-constant string value. As illustrated by this example, TASER can help reduce the false positives of the existing technique by only alerting developers when necessary.

9.5.1 *Limitations*

TASER is affected by the well-known limitations of dynamic analysis, i.e., one can analyze only code that is executed. Therefore, adequate test coverage is essential for effectively extracting taint specifications. Even though

6 https://www.npmjs.com/advisories/27

in our evaluation we do not directly measure or aim to increase coverage for the used test suites, by analyzing several clients of a given library, we increase the chance of observing multiple realistic use cases of the library. Our hypothesis is that these inputs are representative for most of the library usages in the wild. Related work employs similar assumptions [MMT18; MT19].

In this chapter we do not consider implicit flows which we show in Chapter 8 to have limited value for detecting integrity issues in server-side JavaScript. However, future work should evaluate whether this assumption also holds for extracting taint specifications.

In our evaluation, we judge the usefulness of the extracted summaries by showing that they improve an existing static analysis. Similarly to the work of Clapp et al. [CAA15], future work should perform a more extensive set of experiments in which the quality of the extracted specifications is directly evaluated, e.g., by extensively comparing with manually written specifications.

9.6 Conclusion

The massive use of third-party libraries in modern JavaScript web development calls for new techniques to discover security vulnerabilities. Modular static taint analysis is a powerful approach, as demonstrated by the successful commercial tool LGTM, but it critically relies on taint specifications of the libraries being used. Writing such specifications manually is demanding and error-prone, so automated solutions are needed. This chapter presents such a solution. It combines and adapts a number of ideas from previous work, in particular the idea of inferring information flow specifications using dynamic analysis [CAA15], the membrane mechanism [CM10; Mil06], the use of test suites of open-source library clients, and the notion of dynamic access paths [MMT18].

Our implementation and experiments demonstrate that this design is able to automatically detect non-trivial and accurate taint flow specifications in widely used Node.js modules, which enables an existing static analyzer, LGTM, to discover many previously unknown security vulnerabilities. We believe this approach is a promising alternative to the current coarse-grained security tools like `npm audit` that only consider the package dependency structure but completely ignore the dataflow.

Part IV

Security and Privacy Perspectives for Full-Stack JavaScript

10

RELATED WORK

This chapter discusses research work closely related to the current dissertation. We start by introducing work on server-side JavaScript security and on analyzing the implications of (over)using third-party dependencies. We then discuss existing attacks against web applications and empirical studies of the web. Furthermore, we present work on analyzing and improving the performance of JavaScript code, which is critical for preventing availability attacks against web applications. Finally, we discuss different program analyses techniques and tools both aimed at bug detection and at hardening web applications.

10.1 SERVER-SIDE JAVASCRIPT SECURITY

The central thesis of the current dissertation is that the security and privacy of full-stack JavaScript applications requires special treatment from the security community. The main particularity of this type of applications is that they use JavaScript not only on the client-side, but also on the server-side. In recent years, there were several other research efforts, beyond the ones presented in this dissertation, for understanding and improving the security and privacy of server-side JavaScript code. In this section, we discuss this work and relate it to different chapters of the thesis.

DESCRIBING AND DETECTING SECURITY ISSUES The first to assess the security of the NODE.JS platform were Ojamaa and Düüna [OD12]. They identify the most serious threats to the ecosystem: the fragility of the runtime, i.e., the single-threaded nature of the language, the confusing parts of JavaScript, and malicious packages. In the current dissertation we confirm and extend some of these initial assessments (i) by showing that the abuse of error-prone JavaScript APIs, e.g., `eval`, can lead to serious injection vulnerabilities (Chapter 4), (ii) by proposing a novel methodology that enables an attacker to exploit the single-threaded nature of the runtime for performing DoS attacks on live websites (Chapter 5) and (iii) by showing that the continuous rise in the number of dependencies in npm increases the risk of depending on malicious code (Chapter 2). Davis et al. [DKL17;

Dav+18] show that ReDoS vulnerabilities are present in popular modules. In Chapter 5, we take these observations further and show that ReDoS also affects real websites. Injections into Node.js code are known to be exploitable [Sul11] and there is a community-driven effort[1] to identify such problems. In Chapter 4 we perform an in depth study of injection vulnerabilities in npm and show that a high percentage of all libraries are at risk. Gong [Gon18] presents a dynamic analysis system for identifying vulnerable and malicious code in npm. He reports more than 300 previously unknown vulnerabilities, some of which are clearly visible in our analysis of npm vulnerabilities in Section 2.4.3. Pfretzschner et al. [PO17] describe four possible dependency-based attacks that exploit weaknesses, such as global variables or monkeypatching, in Node.js. They implement a detection of such attacks, but they do not find any real-world exploits. Brown et al. [Bro+17] demonstrate that the binding code that is responsible for the interaction between the C++ layer of Node.js and the JavaScript code is prone to a set of security vulnerabilities. In our work, we focus on bugs in the JavaScript code and assume that the lower layers are trustworthy.

TECHNIQUES FOR HARDENING APPLICATIONS Multiple techniques were proposed for increasing the security and privacy of Node.js modules. We discuss below the most important ones and relate them to our defenses presented in Part III of the thesis, i.e., IFlow in Chapter 8, Synode in Chapter 7 and Taser in Chapter 9. NodeSentry [GMP14] is a security architecture for least-privilege integration of Node.js modules. The mechanism uses membranes to enforce security checks at different integration levels. In Chapter 9 we use the same concept for extracting specifications for modules. When compared to NodeSentry, Synode is more powerful since it uses a static analysis to perform fine-grained policy enforcement. BreakApp [Vas+18] proposes OS-level techniques for isolating untrusted third-party code. This technique is much more expensive than Synode for protecting against injection attacks, but it can successfully be deployed against a wider class of attacks. Also compared to Taser, BreakApp is much more coarse-grained, allowing either isolation or complete integration. Node.cure [DWL18] is a technique for protecting against algorithmic complexity attacks, by limiting the amount of time a certain operation can take. Inspired by Synode, AFFOGATO [GHJ18] proposes using a gray-box technique for preventing injection attacks in Node.js. AFFOGATO addresses the precision issues of Synode by proposing a coarse grained

1 https://nodesecurity.io/advisories

technique that compares similarity of strings at sources with the values at sinks.

TECHNIQUES FOR INCREASING SOFTWARE QUALITY While software bugs are not a security issue per se, in certain cases, inadequate software quality can lead to availability issues, e.g., crashes. Since NODE.JS is a single-threaded, event-driven system, concurrency issues can arise due to the order in which events are spawned and processed. Node.fz [DTL17] identifies atomicity and order violations while NodeRacer [EM20] finds data races by analyzing happens-before relations. Mutode [RV18] is a mutation testing tool that uses existing test suites for generating additional tests and, thus, increase test coverage. NoRegrets [MMT18] identifies software evolution problems by finding breaking changes in new releases of npm packages. It uses the test suites of library's clients to infer the semantics of the public API of different versions of a library. When evaluating TASER, we adapt the NoRegrets infrastructure to our orthogonal goal, i.e., extracting taint specifications for libraries. NoRegrets+ [MT19] proposes an optimization of the process that uses model-based techniques and test generation to avoid rerunning the tests for each library release. There are also techniques that address full-stack bugs. For example, Sayagh et al. [SKA17] find configuration errors that arise due to conflicting options in different layers of the stack, while Sahand [AMP16] captures and visualizes asynchronous event interactions between server-side and client-side code. These approaches are similar in spirit with the current dissertation, in the sense that they consider the whole stack, but they address a different problem domain, i.e., software quality instead of security.

10.2 SECURITY IMPLICATIONS OF THIRD-PARTY DEPENDENCIES

In Chapter 2 we perform an in depth study of the npm ecosystem for understanding the risk of depending on malicious or vulnerable code. Below, we discuss similar studies performed for various ecosystems and possibly with different goals in mind, not limiting ourselves to security.

DEPENDENCIES IN THE SERVER-SIDE JAVASCRIPT ECOSYSTEM Dependency management for NODE.JS and its associated risks are a controversial topic, intensely covered in security bulletins and blogs, but also by the research community. After the left-pad incident shook the NODE.JS ecosystem from its foundation, security experts started questioning the use

and role of micro-packages. Abdalkareem et al. [Abd+17] investigate reasons why developers would use trivial packages. They find that developers think that these packages are well implemented and tested and that they increase productivity as the developer does not need to implement such small features herself. Another empirical study of micropackages by Kula et al. [Kul+17] has similar results. They show that micropackages have long dependency chains, something we also discovered in some case studies in Chapter 2. We also show that these packages have a high potential of being the target of an attack as they are dependent on by a lot of packages. Another previously studied topic are breaking changes introduced by dependencies. Bogart et al. [Bog+16] perform a case study interviewing developers about breaking changes in three different ecosystems. They find that the npm community values a fast approach to new releases compared to the other ecosystems. Developers of npm are more willing to adopt breaking changes to fight technical debt. Furthermore, they find that the semantic versioning rules are enforced more over time. Similarly, Decan et al. [DMC17] analyze three package ecosystems, including npm, and evaluate whether dependency constraints and semantic versioning are effective measures for avoiding breaking changes. They find that both these measures are not perfect and that there is a need for better tooling. One such tool can be NoRegrets, a testing technique by Mezzetti et al. [MMT18] which automatically detects whether an update of a package contains a breaking change in the API.

VULNERABILITIES IN SERVER-SIDE DEPENDENCIES As discussed in the previous section, multiple studies identify vulnerabilities or bugs in Node.js libraries: Davis et al. [Dav+18] find denial of service vulnerabilities, Gong [Gon18] finds directory traversals and Wang et al. [Wan+17] analyze concurrency bugs. In this dissertation, we perform two such in-depth studies , i.e., for finding injection and ReDoS vulnerabilities, while in Chapter 2 we analyze the likelihood of a given library to transitively depend on *any* publicly disclosed vulnerability. Furthermore, there are studies that look at how frequent security vulnerabilities are in the npm ecosystem, how fast packages fix these and how fast dependent packages upgrade to a non-vulnerable version. Chatzidimitriou et al. [Cha+18] build an infrastructure to measure the quality of the npm ecosystem and to detect publicly disclosed vulnerabilities in package dependencies. Decan et al. [DMC18] perform a similar study, but they investigate the evolution of vulnerabilities over time. They find that more than half of the dependent

packages are still affected by a vulnerability after the fix is released. We show that the problem is even more serious because for more than half of the npm packages there is no patch available.

CLIENT-SIDE (JAVASCRIPT) DEPENDENCIES A central theme of the current dissertation is the excessive code reuse in the JavaScript ecosystem. This is a well known problem and it was extensively studied for traditional web applications, i.e., in which only the front-end of the application is written in JavaScript. Below, we present the most important studies that analyze the excessive dependencies in client-side code. Nikiforakis et al. [Nik+12] present a study of remote inclusion of JavaScript libraries in the most popular 10,000 websites. They show that an average website in their data set adds between 1.5 and 2 new dependencies per year. Similar to our work in Chapter 2, they then discuss several threat models and attacks that can occur in this tightly connected ecosystem. Lauinger et al. [Lau+17] study the inclusion of libraries with known vulnerabilities in both popular and average websites. They show that 37% of the websites in their data set include at least one vulnerable library. This number is surprisingly close to the reach we observe in npm for the vulnerable code. However, one should take both these results with a grain of salt since inclusion of vulnerable libraries does not necessary lead to a security problem if the library is used in a safe way. Libert et al. [Lib15] perform an HTTP-level analysis of third-party resource inclusions, i.e., dependencies. They conclude that nine in ten websites leak data to third-parties and that six in ten trigger the creation of third-party cookies. Snyder et al. [STK17] discuss the cost-benefit analysis of depending on different web APIs. That is, they show that certain APIs are the culprit of many security vulnerabilities, while only being used by a small number of websites.

STUDIES OF OTHER SOFTWARE ECOSYSTEMS Software ecosystem research has been rapidly growing in the last years. Manikas [Man16] surveys the related work and observes a maturing field at the intersection of multiple other research areas. Nevertheless, he identifies a set of challenges, for example, the problem of generalizing specific ecosystem research to other ecosystems or the lack of theories specific to software ecosystems. Serebrenik et al. [SM15] perform a meta-analysis of the difficult tasks in software ecosystem research and identify six types of challenges. For example, how to scale the analysis to the massive amount of data, how to research the quality and evolution of the ecosystem and how to dedicate

more attention to comparative studies. Mens [Men16] further takes a socio-technical view on software maintenance and evolution. He argues that future research needs to study both the technical and the social dimensions of the ecosystem. Our study in Chapter 2 follows this recommendation as it not only looks at the influence of a package on the npm ecosystem, but also at the influence of the maintainers. Several related work advocates metrics borrowed from other fields. For example, Lertwittayatrai et al. [Ler+17] use network analysis techniques to study the topology of the JavaScript package ecosystem and to extract insights about dependencies and their relations. Another study by Kabbedijk et al. [KJ11] looks at the social aspect of the Ruby software ecosystem by identifying different roles maintainers have in the ecosystem, depending on the number of developers they cooperate with and on the popularity of their packages. Similarly to our preliminary study of dependences on injection APIs in Chapter 4 and our ecosystem study in Chapter 2, Mastrangelo et al. analyze the usage of unsafe API in Java [Mas+15]. Other analyses for Java [Rei+16] and Android applications [BBD16; Li+17] focus on security risks due to libraries, which shares with this dissertation the idea to consider third-party code as a potential security threat. Overall, the research field is rising with a lot of studied software ecosystems in addition to the very popular ones, such as JavaScript, which is the focus of this dissertation.

ECOSYSTEM EVOLUTION Studying the evolution of an ecosystem shows how fast it grows and whether developers still contribute to it. Wittern et al. [WSR16] study the whole JavaScript ecosystem, including GitHub and npm until September 2015. They focus on dependencies, the popularity of packages and version numbering. They find that the ecosystem is steadily growing and exhibiting a similar effect to a power law distribution as only a quarter of packages is dependent upon. Comparing these numbers with our results in Chapter 2, we see a continuous near-exponential growth in the number of released packages and that only 20% of all packages are dependent upon. A similar study by Kikas et al. [Kik+17] that includes the JavaScript ecosystem collects data until May 2016 and focuses on the evolution of dependencies and the vulnerability of the dependency network. They confirm the same general growth as the previous study. Furthermore, they find certain packages that have a high impact with up to 30% of other packages and applications affected. In Chapter 2, we provide an up-to-date view for these studies and we additionally look at the evolution of maintainers as they are a possible vulnerability of the ecosystem. The de-

pendency network evolution was also studied for other ecosystems. Decan et al. [DMG19] compare the evolution of seven different package managers focusing on the dependency network. Npm is the largest ecosystem in their comparison and they discover that dependencies are frequently used in all these ecosystems with similar connectedness between packages. Bloemen et al. [Blo+14a] look at software package dependencies in the Linux distribution *Gentoo* where they use cluster analysis to explore different categories of software. German et al. [GAH13] study the dependency network of the *R* language and the community around its user-contributed packages. Bavota et al. [Bav+13] analyze the large Apache ecosystem of Java libraries where they find that while the number of projects grows linearly, the number of dependencies between them grows exponentially. Comparing this to the npm ecosystem, we find the number of packages to grow super-linearly while the average number of dependencies between them grows linearly.

VULNERABILITY NOTIFICATION Several researchers document the difficulty of notifying the maintainers of websites or open-source projects about security bugs in software [Lek+15; Sto+18; Sto+16]. In our work, we had inconsistent experiences. On one hand, when reporting injection vulnerabilities in Chapter 4, we had a hard time convincing package maintainers to fix their code or even to reply to our requests. Similarly, when reporting ReDoS vulnerabilities in Chapter 5, we had a hard time getting the fixes deployed, but we experienced better interactions with the maintainers through the Node.js security team. On the other hand, when reporting leaky images vulnerabilities in popular websites in Chapter 6, we experienced quick and helpful responses by all websites we contacted, with an initial response within less than a week. One reason for this difference may be that in the latter case we used the bug bounty channels provided by the websites to report the problems [FAW13; ZGL15]. Another reason may be that in the former case we interacted with open source package maintainers, while in the latter we communicated our findings to major, multi-national, technology companies.

10.3 ATTACKS AGAINST WEB APPLICATIONS

Web application attacks represent a vast and maturing field which is impossible to survey in the current dissertation. Therefore, below we limit

our discussion to work that is closely related to our novel leaky images attack, discussed in Chapter 6.

SAME ORIGIN EXCEPTIONS Previous work shows risks associated with images on the web, such as image-based fingerprinting of browser extensions [SAS17], malicious JavaScript code embedded in SVGs [Hei+11], and leaking sensitive information, such as the gender or the location of a user uploading an image [CS16b]. This dissertation introduces a new risk: privacy leaks due to shared images. Lekies et al. [Lek+15] describe privacy leaks resulting from dynamically generated JavaScript. The source of this problem is the same as for leaky images: Both JavaScript code and images are excepted from the same-origin policy. While privacy leaks in dynamic JavaScript reveal confidential information about the user, such as credentials, leaky images allow for tracking specific users on third-party websites. Heiderich et al. [Hei+14] introduce a scriptless, CSS-based web attacks. The HTML-only variant of leaky images does not rely on CSS and also differs in the kinds of leaked information: while the attack by Heiderich et al. leaks content of the current website, our attacks leak the identity of the user.

SOCIAL MEDIA ATTACKS Wondracek et al. [Won+10] present a privacy leak in social networks related to our group attack in Chapter 6.3.3. In their work, the attacker neither has control over the group structure nor can she easily track individuals. A more recent attack [Su+17] deanonymizes social media users by correlating links on their profiles with browsing histories. In contrast, our leaky image attack does not require such histories. Another recent attack [Ven+18] retrieves sensitive information of social media accounts using the advertisement API provided by a social network. However, their attack cannot be used to track users on third-party websites.

BROWSER FINGERPRINTING Browser fingerprinting is a widely deployed [Aca+13; Aca+14; Nik+13] persistent tracking mechanism. Various APIs have been proposed for fingerprinting: user agent and fonts [Eck10], canvas [CLW17; MS12], ad blocker usage, and WebGL Renderer [LRB16]. Empirical studies [Eck10; LRB16] suggest that these technique have enough entropy to identify most of the users, or at least, to place a user in a small set of possible users, sometimes even across browsers [CLW17]. The leaky image attack is complementary to fingerprinting, as discussed in detail in Section 6.3.6.

AUTHENTICATED THIRD-PARTY REQUESTS Cross-Site Request Forgery (CSRF) is similar in spirit to leaky image attacks: Both rely on the fact that browsers send cookies with third-party requests. For CSRF, this behavior results in an unauthorized action on a third-party website, whereas for leaky images, it results in deanonymizing the user. Existing techniques for defending [BJM08] and detecting [Pel+17] CSRF partially address but do not fully solve the problem of leaky images (Section 6.5). Another web tracking mechanism is through third-party requests, such as tracking pixels. Mayer and Mitchell [MM12] describe the tracking ecosystem and the privacy costs associated with these practices. Lerner et al. [Ler+16] show how tracking in popular websites evolves over time. Several other studies [Cah+16; EN16; Eng+15; RKW12; Tra+12; Yu+16] present a snapshot of the third-party tracking on the web at various moments in time. One of the recurring conclusion of these studies is that few big players can track most of the traffic on the Internet. We present the first image-based attack that allows a less powerful attacker to deanonymize visitors of a website.

ADVANCED PERSISTENT THREATS Targeted attacks or advanced persistent threats [Mar+14; SE13] are an increasingly popular class of cybersecurity incidents. Known attacks include spear phishing attacks [Blo+14b] and targeted malware campaigns [Har+14; Mar+14]. Leaky images adds a privacy-related attack to the set of existing targeted attacks.

BEYOND JAVASCRIPT ATTACKS User privacy can also be impacted by security issues in browsers, such as JavaScript bindings bugs [Bro+17], micro-architectural bugs [Koc+19], and insufficient isolation of web content [Jia+16]. Neither of these studies explores privacy leaks caused by authenticated cross-origin image requests. Van Goethem et al. [GJN15] propose the use of timing channels for estimating the file size of a cross-origin resource. One could combine leaky images with such a channel to check if a privately shared image is accessible for a particular user, enabling the use of leaky images even if the browser would block cross-origin image requests. One difference between our attack and theirs is that leaky images provide 100% certainty that a victim has visited a website, which a probabilistic timing channel cannot provide.

In the current dissertation, we perform various empirical studies, both of the npm ecosystem and of live websites. This type of work is sometimes referred as *measurement studies* in the security community and it comprises a huge body of recent work. Below, we limit the discussion to two types of studies: studies that directly relate to our obfuscation and minification work in Chapter 3 and general web security studies that relate to our work on detecting vulnerabilities in live websites, i.e., both the ReDoS work in Chapter 5 and the leaky images study in Chapter 6.

OBFUSCATION STUDIES Ceccato et al. [Cec+15] analyze the impact of different obfuscation patterns on Java code in terms of the difference between the obfuscated and the original code with the help of metrics. Wang et al. [Wan+18] examine the characteristics of obfuscated iOS Apps, the popularity of different obfuscation patterns, and the difficulty of reverse engineering the obfuscated apps. Hammad et al. [HGM17] perform a study concerning the effect of obfuscated Android apps on anti-malware products. They find that the performance of most anti-malware products is significantly impacted by obfuscated apps. Moreover, they report that the tools used for obfuscating the apps frequently result in corrupt apps. Depending on the tool, only 0% to 62% of all 250 apps are runnable after the obfuscation is applied. We observe a similar outcome in the case of JavaScript obfuscation. Visaggio et al. [VPC13] compare obfuscated and regular JavaScript code using several metrics, including n-gram, entropy, and word size. They find that the two kinds of code differ from each other, in particular, when combining multiple metrics. Xu et al. [XZZ12] study 510 samples of malicious JavaScript code, its use of obfuscation, and how obfuscation influences whether an anti-virus checker detects the code as malicious. Fass et al. [Fas+18] propose using a random forest classifier based on code structure for detecting malicious JavaScript instances. In contrast, our study in Chapter 3 considers many more scripts, goes beyond obfuscated code, and addresses different research questions. As discussed below, there have been various JavaScript and web security related studies, but none focuses on obfuscation and minification in the web, outside of malicious code. Therefore, in Chapter 3, we conduct the first comprehensive study examining the prevalence, the performance, and the validity of obfuscated and minified JavaScript code in the web.

WEB SECURITY STUDIES Several web security measurements studies of live websites were performed recently: on the prevalence of the `eval` function [Ric+11; YW09], on trust relationships between websites that include remote libraries and their corresponding library providers [Nik+12], on the communication between websites and embedded frames with third-party content [SS13], on cookie stealing [RJLS10; SPK16], on credentials theft [AHS17], on outdated libraries in the web [Lau+17], on deployment of CSP policies [Wei+16a] and on XSS vulnerabilities [LSJ13; Mel+18]. To the best of our knowledge, the current dissertation is the first to focus on server-side JavaScript issues in live websites and on ReDoS vulnerabilities. Similarly, several measurement studies investigate tracking practices on the web: by examining third-party requests [Cah+16; EN16; Eng+15; RKW12; Tra+12; Yu+16] or by analyzing the prevalence of browser fingerprinting [Aca+14; Nik+13]. We are the first to discuss tracking users using authenticated image requests, i.e., leaky images. However, the prevalence of this type of tracking in the wild is unknown.

10.5 PERFORMANCE OF JAVASCRIPT CODE AND DOS ATTACKS

In Chapter 5, we study in depth ReDoS vulnerabilities, an algorithmic complexity attack that affects both open-source libraries and live websites. Below, we discuss related work on detecting such performance problems, both in JavaScript and in other programming languages.

ANALYSIS OF REDOS VULNERABILITIES Prior work analyzes the worst case matching time of regular expressions [BI16; BDM14; KRT13; Wei+16b]. Most of this work assumes backtracking-style matching and analyzes regular expressions in isolation, ignoring whether attacker-controlled inputs reach it. Recent work by Wüstholz et al. [Wüs+17] considers this aspect. They combine static analysis and exploit generation to find 41 vulnerabilities in Java software. Our work differs in three ways: (i) we analyze JavaScript ReDoS, which is more serious than Java ReDoS, (ii) we detect vulnerabilities in real-world websites whose source code is not available for analysis, and (iii) we uncover ReDoS vulnerabilities containing advanced features, e.g. lookahead, that are not supported by any of the previous work. A study performed concurrently with ours considers ReDoS vulnerabilities in the npm ecosystem and confirms that ReDoS is a serious threat for JavaScript code [Dav+18].

REGULAR EXPRESSIONS Regular expressions are often used for sanitizers and XSS filters. Bates et al. [BBJ10] show that XSS filters are often slow, incorrect, and sometimes even introduce new vulnerabilities. Hooimeijer et al. [Hoo+11] show that supposedly equivalent implementations of sanitizers differ. A study by Chapman et al. [CS16a] shows that developers have difficulties in composing and reading regular expressions. We are the first to analyze the impact of this problem on real-world websites. To avoid mistakes in regular expressions, developers may synthesize instead of writing them [Bar+14; Bar+16].

ALGORITHMIC COMPLEXITY ATTACKS Differences between average and worst case performance are the basis of algorithmic complexity attacks. Crosby and Wallach [CW03] analyze vulnerabilities due to the performance of hash tables and binary trees, while Dietrich et al. [Die+17] study serialization-related attacks. Wise [BJS09], SlowFuzz [Pet+17], and PerfSyn [TPG18] generate inputs to trigger unexpectedly high complexity.

RESOURCE EXHAUSTION ATTACKS SAFER [Cha+09] statically detects CPU and stack exhaustion vulnerabilities involving recursive calls and loops. Huang et al. [Hua+15] study blocking operations in the Android system that can force the OS to reboot when called multiple times. Shan et al. [SWP17] consider attacks on n-tier web applications and model them using a queueing network model.

TESTING REGULAR EXPRESSIONS The problem of generating inputs for regular expressions is also investigated from a software testing perspective [LK16; Li+09; SMS12; VHT10]. In contrast to our work, these techniques aim at maximizing coverage or finding bugs in the implementation.

PERFORMANCE OF JAVASCRIPT ReDoS vulnerabilities are a kind of performance problem. Such problems are worth fixing independent of their exploitability in a denial of service attack, e.g., to prevent websites from being perceived as slow and unresponsive. Existing work has studied JavaScript performance issues [SP16] and proposed profiling techniques to identify them [GPS15; Jen+15; Pra+14]. Studying the exploitability of other performance issues beyond ReDoS is a promising direction for future work.

This dissertation heavily builds on program analysis techniques, e.g., information flow control, membranes, data flow analysis, etc. In the remaining part of this chapter we discuss closely related work in this area. Once again, we do not aim to perform a comprehensive literature review due to the vastness of the field [And+17], but instead, describe the most influential work. In this section, we limit the discussion to high-level research directions in program analysis, while in the following sections we present techniques for hardening application security.

PRAGMATIC PROGRAM ANALYSIS The dynamic and reflective nature of JavaScript makes it difficult to construct sound, whole-program static analyses that scale to large real-world applications [AM14; Fel+13; JMT09; KM19; NHG19; SL19; SP18]. For that reason, much research has been devoted to constructing more pragmatic bug-detection tools [And+17; CN15; Gon+15; Hed+17; Hed+14; Kar+18; PSS15; Sch+13; Sel+18]. We followed in this tradition when designing both SYNODE and TASER. Some frameworks facilitate the implementation of dynamic analyses [Sen+13; Sun+18], including NodeProf [Sun+18] that TASER builds upon.

NODE.JS TECHNIQUES Madsen et al. [MTL15a] enhance the call graph construction for NODE.JS applications with event-based dependencies. The static analysis component of SYNODE is intra-procedural, but it could benefit from integrating such an inter-procedural approach, which may further reduce the false positive rate. Nielsen et al. [NHG19] and Madsen et al. [Mad+16] propose using feedback-driven program analysis to avoid analyzing less interesting parts of the application. Most of our program analyses, e.g., SYNODE or TASER, could benefit from such optimizations.

MEMBRANES The membrane pattern, introduced by Miller [Mil06], has been applied in several settings [CM10; GMP14; KT15; MMT18; Mil06; MT19]. The idea is to separate two object graphs, such that operations taking place on the boundary between the graphs can be captured and potentially modified. TASER uses membranes at the boundary between a module and a client, and between different modules, to capture taint flows between them.

BUG DETECTION Our work relates to program analyses for detecting bugs in JavaScript code, e.g., finding conflicts between libraries [PDP18], type inconsistencies [PSS15], naming errors [Liu+16], code quality violations [Gon+15] or data races [MTL15b]. However, none of these techniques target security and privacy issues which are at the core of this dissertation.

MACHINE LEANING AND PROGRAM ANALYSIS In Chapter 3, we propose using a machine learning model for identifying obfuscated and minified code. This relates to a growing body of recent work on applying machine learning to program analysis: for interring types [Hel+18; MPP19], for identifying concurrency issues [HP18] or for finding bugs [Bad+19; HP19; PS18]. For a comprehensive survey of this area we point the reader to the work of Allamanis et al. [All+18]. Below, we discuss existing work on detecting obfuscated and minified code, mostly by using machine learning techniques, and discuss how it relates to the classifier we propose in Chapter 3.

MACHINE LEARNING FOR OBFUSCATION DETECTION Tellenbach et al. [TPR16] propose multiple classifiers for detecting obfuscation. However, they manually specify features of code files, such as the Shannon entropy or the number of characters per line. Similarly, Likarish et al. [LJJ09] manually extract 60 features from code and propose four different classifiers. In contrast to both, our AST-based classifier does not require any feature engineering. Wang et al. [WCW16] present a neural network-based classifier for detecting malicious JavaScript code. Since attackers often hide the malicious intent of their scripts using obfuscation, a significant proportion of the dataset used for training their classifier consists of obfuscated code. Comparable to our approach, instead of defining explicit features, they learn the features automatically from the code with the help of multiple layers of stacked denoising autoencoders in the neural network. They transform the code files by replacing each character with a unique binary vector. The vectors require 20,000 dimensions to represent all characters of the dataset. Therefore, Wang et al. reduce the dimensionality to 480 using a dimensionality reduction algorithm. In contrast, we learn a vector representation for AST nodes which only requires 45 dimensions. Al-Taharwa et al. [AT+15] present a Bayesian-based obfuscation detector aimed at manually performed obfuscation. In contrast, we consider automatically transformed code. Kaplan et al. [Kap+11] and Curtsinger et al. [Cur+11] extract features from AST nodes while preserving the context of nodes. Their clas-

sifiers specialize on obfuscated and on malicious code, respectively. The context-based feature extraction mechanism is comparable to the concept we follow by processing entire ASTs of code files to preserve the context of all AST nodes. However, all three approaches limit the number of allowed contexts and reduce the number of features by applying feature selection. In our case, we do not discard any information from the ASTs so that the neural network can extract the most descriptive features. There are further static classifier-based approaches which are less related to our approach. Jodavi et al. [JAP15] address the detection of obfuscation with an ensemble of multiple one-class SVM classifiers which are trained to recognize non-obfuscated code using a set of structural and lexical features, such as the number of dynamic code evaluations and the maximum entropy of strings.

OTHER TECHNIQUES FOR DETECTING OBFUSCATED CODE Certain researchers address the problem of detecting obfuscated code by using techniques other than machine learning. Xu et al. [XZZ13] propose an approach to detect malicious obfuscated code using both static and dynamic analysis. They mostly focus on obfuscation techniques which require the usage of the `eval` function, the `unescape` function, or other related functions. Using static analysis, they gather information about function definitions and invocations in the code. At runtime, they examine if new function definitions and invocations are present to reveal the potentially malicious part of the code. In contrast, we detect if code is obfuscated without assuming a malicious intent because obfuscation allows for different goals, e.g., the protection of intellectual property of code. Furthermore, we include all techniques offered by the obfuscation tools we use for the study instead of focusing only on `eval` based obfuscation. As discovered in Section 3.2.2.2, most JavaScript obfuscators do not implement `eval` based techniques. Ceccato et al. [Cec+16] propose using out-of-the-box obfuscators for hindering portability of attacks. We also use readily available obfuscators, but for generating training data, not for software diversification. Deobfuscation is the process of reverse engineering obfuscated code. Techniques for this purpose include learning-based approaches [BPS18; RVK15] and semantics-preserving rewriting of the obfuscated code [LD12; Yak+16]. This work is complementary to our work and can be applied after detection of obfuscated code.

In Part III of the thesis we introduce three techniques for increasing security and privacy of web applications: Synode in Chapter 7, iFlow in Chapter 8, and Taser in Chapter 9. These techniques relate to a growing body of work on hardening web applications, both by the research community and by practitioners. Below, we discuss the closest related work, omitting the information flow techniques which are discussed in Section 10.8.

COARSE-GRAINED ALERTS Some tools, most prominently `npm audit`[2] and Snyk[3], warn developers about known vulnerabilities in any of their dependencies. As discussed by Lauinger et al. [Lau+17], an important limitation is that such tools do not analyze how dependencies are used, and will warn even about vulnerabilities in code that a client does not use, or not use in a vulnerable way. A more precise analysis, e.g., based on specifications inferred by Taser, avoids the inevitable false positives caused by coarse-grained alerts.

PREVENTING XSS ATTACKS Blueprint [LV09] prevents XSS attacks by enforcing that the client-side DOM resembles a parse tree learned at the server-side. Their work shares with Synode the idea of comparing data prone to injections to a tree-based template. Our work differs by learning templates statically and by focusing on command injections in Node.js code. Stock et al. [Sto+14] study DOM-based XSS injections and propose dynamic taint tracking to prevent them. Similar to Synode, their prevention policy is grammar-based. However, their strict policy to reject any tainted data that influences JavaScript code except for literals and JSON would break many uses of `eval` found during our study. Another difference is that we avoid taint tracking by statically computing sink-specific templates. Defenses against XSS attacks [Mit+16; SBS15] use signature-based whitelisting to reject scripts not originating from the website creator. SICILIAN [SBS15] uses an AST-based signature; nsign [Mit+16] creates signatures from script-dependent elements and context-based information. Both rely on a training phase to discover valid signatures. Synode also uses templates as a white-listing mechanism. However, we do not rely on testing to collect these templates but compute them statically. As we show in Section 7.5 , there may be hundreds or even thousands of paths that

2 https://docs.npmjs.com/cli/audit
3 https://snyk.io/

reach an injection API call site, i.e., constructing valid signatures for every path is infeasible.

REWRITING-BASED TECHNIQUES CSPAutoGen [Pan+16] presents an automatic way to generate CSP policies on the server-side in order to protect against illegitimate script execution on the client-side. It uses gASTs, partial ASTs similar in structure with SYNODE's, but different in many ways. First of all, gASTs are created during a training session, which limits the approach to behavior observed during the training phase. gASTs also differ from our partial ASTs in the way they are enforced: gASTs are synthesized into a JavaScript function that replaces the actual call to the sink. Such a step cannot be easily implemented for sinks other than `eval` since these sinks call outside the JavaScript world. For example, to refactor a call to `exec` that uses the `awk` system utility on Linux, we would need to completely rewrite `awk` in JavaScript. Several approaches rewrite JavaScript code to enforce security policies. Yu et al. [Yu+07] propose a rewriting technique based on edit automata that replaces or modifies particular calls. Gatekeeper [GL09] is a static analysis for a JavaScript subset that enforces security and reliability policies. Instead of conservatively preventing all possibly insecure behavior, SYNODE defers checks to runtime when hitting limitations of purely static analysis. Other techniques [JJM12; Mea+12] replace `eval` calls with simpler, faster, and safer alternatives. Their main goal is to enable more precise static analysis; SYNODE focuses on preventing injections at runtime.

PREVENTING INJECTIONS VIA SANITIZATION CSAS [SSS11] uses a type system to insert runtime checks that prevent injections into template-based code generators. Livshits et al. [LC13] propose to automatically place sanitizers into .NET server applications. Similar to SYNODE, these approaches at first statically address some code locations and use runtime mechanisms only for the remaining ones. CSAS differs from our work by checking code generators instead of final code. The approach in [LC13] addresses the problem of placing generic sanitizers, whereas we insert runtime checks specific to injection call sites.

RUNTIME DEFENSES AGAINST INJECTION ATTACKS There are several purely dynamic approaches to prevent injections. XSS-Guard [BV08] modifies server applications to compute a shadow response along each actual response and compares both responses to detect unexpected, injected con-

tent. Instead of comparing two strings with each other, Synode compares runtime strings against statically extracted templates. ScriptGard [SML11] learns during a training phase which sanitizers to use for particular program paths and detects incorrect sanitization by comparing executions against the behavior seen during training. Their approach is limited by the executions observed during training and needs to check all execution paths, whereas Synode statically identifies some locations as safe. Su and Wassermann [SW06] formalize the problem of command injection attacks and propose grammar-based runtime prevention. Their work shares the idea to reject runtime values based on a grammar that defines which parts of a string may be influenced by attacker-controlled values. Their analysis tracks input data with special marker characters, which may get lost on string operations, such as `substring`, leading to missed injections. Instead, Synode does not need to track input values through the program. Buehrer et al. [BWS05] take a similar approach to mitigate SQL injections. They construct two parse trees at runtime, one representing the developers intentions only and one including the user input. They use these trees to ensure that the user input contains only literals. Their approach is purely dynamic and employs markers for tracking user input, similar to Su and Wassermann [SW06]. Ray and Ligatti [RL12] propose a novel formulation of command injections that requires dynamic taint tracking and a set of trusted inputs. For Node.js libraries, example inputs are rarely available.

CONSTRAINT SOLVERS Constraint-based static string analysis, e.g., Z3-str [ZZG13] is a more heavy-weighted alternative to our static analysis in Synode. Even though such techniques have the potential of producing more precise templates, we opted for efficiency when designing Synode, enabling us to easily apply the analysis to thousands of npm modules.

EXPLOIT GENERATION Wassermann et al. address the problem of finding inputs that trigger SQL injections [Was+08] and that exploit XSS vulnerabilities [WS08] in PHP code. Ardilla [Kie+09] finds and exploits injection vulnerabilities in PHP through a combination of taint analysis and test generation. Instead of triggering attacks, the enforcement mechanisms in this dissertation address the problem of preventing attacks.

Work	Analysis	Explicit	Obs.	Hidden
Vogt et al. [Vog+07]	dynamic	✓	✓	-
Jang et al. [RJLS10]	hybrid	✓	✓	-
Chugh et al. [Chu+09]	hybrid	✓	✓	✓
Tripp et al. [TFP14]	hybrid	✓	-	-
Chudnov & Naumann [CN15]	dynamic	✓	✓	NSU
Hedin et al. [Hed+14]	dynamic	✓	✓	NSU
Bichhawat et al. [Bic+17]	dynamic	✓	✓	PU
Kerschbaumer et al. [Ker+13]	dynamic	✓	✓	-
Bauer et al. [Bau+15]	dynamic	✓	-	-
De Groef et al. [DG+12]	dynamic	MOD	MOD	MOD
Austin & Flanagan [AF12]	dynamic	MOD	MOD	MOD

TABLE 10.1: JavaScript information flow analyses and the flows they support: ✓ = considers this flow, - = does not consider this flow, NSU and PU = may abort due to No Sensitive Upgrade (NSU) or Permissive Upgrade (PU) checks, respectively, and MOD = may modify program behavior.

10.8 INFORMATION FLOW ANALYSIS

In this section we discuss information flow control, a program analysis technique that is at the core of our methodologies in Chapter 8 and Chapter 9. Denning and Denning pioneered the development and formal description of static information flow analyses [Den76; DD77]. Fenton studies purely dynamic information flow monitors [Fen74]. A huge body of work has been created during the years to refine Dennings' and Fenton's ideas and to adapt them to various languages. Table 10.1 presents some of the more recent tools and shows what kinds of flows they consider. We look at three types of information flow as introduced in Chapter 8: explicit, observable implicit and hidden implicit flows. Many analyses, including TASER, consider only explicit flows [SAB10]. Among the analyses that consider implicit flows, the majority stop or modify the program as soon as a hidden flow occurs.

INFORMATION FLOW ANALYSIS FOR JAVASCRIPT Chugh et al. propose a static-dynamic analysis that reports flows from code given to `eval` to sensitive locations, such as the location bar of a site [Chu+09]. Austin and Flanagan address the problem of hidden implicit flows [AF09; AF10], as discussed in detail in Section 8.2. Hedin and Sabelfeld propose a dynamic analysis that implements the No Sensitive Upgrade (NSU) strategy,

discussed in Section 8.3.1, for a subset of JavaScript [HS12]. They develop JSFlow, which supports the full JavaScript language, but it requires inserting upgrade statements manually [Hed+14]. Birgisson et al. propose to automatically insert upgrade statements [BHS12] by iteratively executing tests under the NSU monitor. Their approach is implemented for a JavaScript-like language, whereas IFLOW supports the full JavaScript language. Our monitor in Chapter 8 implements the Permissive Upgrade (PU) strategy to insert upgrade statements, which reduces the number of upgrade statements and increases permissiveness. Bichhawat et al. propose a variant of PU, where the program is terminated whenever a partially leaked value may flow into the heap [Bic+14]. A WebKit-based browser by Kerschbaumer et al. [Ker+13] balances performance and permissiveness by probabilistically switching between taint tracking and observable tracking and deploys crowdsourcing techniques to discover information flow violations by Alexa Top 500 pages.

OTHER WORK ON INFORMATION FLOW ANALYSIS Balliu et al. study a family of information flow trackers for different kinds of flows and propose security conditions to evaluate their soundness [BSS17]. In Chapter 8, we borrow their conditions to prove the soundness of our monitor for NanoJS. Bao et al. show that considering implicit flows can cause a significant amount of false positives and propose a criterion to determine a subset of all conditionals to consider [Bao+10]. Chandra and Franz propose a VM-based analysis for Java that combines a conservative static analysis with a dynamic analysis to track all three kinds of flows considered in Chapter 8 [CF07]. Dytan is a dynamic information flow analysis for binaries that supports both explicit and observable implicit flows [CLO07]. Myers and Liskov introduce Jif, a language for specifying and statically enforcing security policies for Java programs [ML00]. A survey by Sabelfeld and Myers provides an overview of further static approaches [SM03].

APPLICATIONS OF INFORMATION FLOW ANALYSIS Information flow analysis is widely used to discover potential vulnerabilities. All approaches we are aware of consider only a subset of the three kinds of flows. Flax uses taint analysis to find incomplete or missing input validation and generates attacks that try to exploit the potential vulnerabilities [Sax+10]. Lekies et al. [LSJ13] and Melicher et al. [Mel+18] propose a similar approach to detect DOM-based XSS vulnerabilities. Jang et al. analyze various web sites with information flow policies targeted at common privacy leaks and at-

tack vectors, such as cookie stealing and history sniffing [RJLS10]. Their analysis considers observable implicit flows, but not hidden implicit flows. Sabre analyzes flows inside browser extensions to discover malicious extensions [DG09]. Taint analysis [CLO07], a light version of information flow control that only considers explicit flows, has been used for checking security properties [Arz+14; Gua+11; NS05; Tri+09], and for other analysis problems [GLR09; HZ16]. In particular, there are both static [NHG19] and dynamic [Kar+18; LSJ13] taint analyses for JavaScript. Taint specifications inferred with an Taser-like approach could in principle be plugged into any static taint analysis that involves third-party modules. To the best of our knowledge, we are the first to present such an approach for static taint analysis for JavaScript. To facilitate the use of taint analysis for checking security properties, some work proposes to infer which functions to consider as sources, sinks, and sanitizers [Chi+19; RAB14]. In contrast, Taser infers specifications that summarize flows through entire third-party modules.

SPECIFICATIONS OF LIBRARIES AND FRAMEWORKS The idea of using pre-generated specifications to aid static analysis of library and framework code has been pursued previously [AB16; Bas+18; CAA15; HSC15; PJR19]. The only work other than Taser that uses a dynamic analysis to infer taint specifications is the technique by Clapp et al. [CAA15], which infers specifications for the Android SDK. Our work differs in multiple ways. First, we introduce the idea of membrane-based, multi-module analysis, allowing Taser to infer specifications for all modules used directly or indirectly by a client. In contrast, Clapp et al. [CAA15] infer specifications from a client's usage of a single framework. Second, we use a fine-grained specification mechanism that can track flows at the level of individual properties and can express flows via callbacks, while their specifications are coarse-grained, i.e., only tracking flows between parameters and return values. Finally, our analysis accounts for the dynamic nature of JavaScript, e.g., using the star expression ($\star$) as described in Section 9.2.2.

STUDIES OF INFORMATION FLOW King et al. [Kin+08] share our goal from Chapter 8 of understanding practical trade-offs between explicit and implicit flows. They empirically study implicit flows detected by a static analysis in six Java-based implementations of authentication and cryptographic functions. They report that most of the reported policy violations are false positives, mostly due to conservative handling of exceptions. Our work focuses on dynamic analysis for JavaScript-based implementations,

which gives rise to a class of observable secrecy monitors that is not relevant in a static setting. Another empirical study of information flows is by Masri and Podgurski [MP09]. Their work studies how the length of flows (measured as the length of the static dependence chain), the strength of flows (measured based on entropy and correlations), and different kinds of information flows (explicit and observable implicit) relate to each other. Similar to our methodology in Chapter 8, Masri and Podgurski target dynamic analysis. Our work differs by addressing different research questions, a different language, and by considering hidden implicit flows.

11

CONCLUSIONS

In this final chapter, we recapitulate the high-level contributions of this dissertation and we highlight the most important future work directions.

11.1 SUMMARY OF CONTRIBUTIONS

In the first part of this thesis, we provide empirical evidence that code transformations are widespread on the web and that excessive code reuse in the server-side ecosystem increases the risk of depending on malicious or vulnerable code. In the second part, we present novel vulnerabilities and attacks. First, we study in detail two security problems that are aggravated by the new threat model for JavaScript: injection and regular expression denial of service (ReDoS) vulnerabilities. We show that these problems are widespread in server-side library code and that a motivated attacker can exploit such vulnerabilities in live, real-world websites. Second, we present a novel targeted privacy attack against web applications, called leaky images. We show that an attacker can mount such an attack against users of popular websites and that automatically detecting vulnerable websites requires full-stack support. In the third part of the thesis, we propose novel techniques or improvements to existing techniques for taking into account the identified particularities and vulnerabilities. First, we propose Synode, a tool for defending against injection attacks at runtime and show that a lightweight data-flow analysis is enough for handling most identified injection vulnerabilities. Second, we show that taint analysis suffices for most of the vulnerabilities in server-side JavaScript code. However, when dealing with client-side confidentiality issues or when the analysis aims to consider malicious code, implicit flows should be taken into account as well. Finally, we present Taser, a technique for dynamically extracting taint specifications for JavaScript libraries. We show that this approach improves the static analysis of server-side JavaScript.

As discussed in Chapter 10, studying the security and privacy of full-stack JavaScript web applications is a new research direction with multiple recent contributions. Below, we highlight a set of future work ideas directly connected with this dissertation.

SECURITY IMPLICATIONS OF JAVASCRIPT EVERYWHERE In this dissertation, we discuss in detail the security implications imposed by the new threat model introduced by server-side JavaScript. We show that developers have a hard time accounting for the new security assumptions, such as the lack of sandbox. Recently, there is an increasing tendency to run JavaScript in various other use cases, beyond web applications: for mobile development, for IoT applications or even for developing standalone desktop applications. Naturally, these new use cases come with their unique threat models and, thus, reusing code developed for web applications in these new settings may lead to surprising results. We believe that future work should explore the implications of running legacy code on the plethora of emerging JavaScript platforms.

FULL-STACK JAVASCRIPT PROGRAM ANALYSIS Detecting certain web vulnerabilities require the analysis of full-stack information flows, e.g., privacy leaks (including leaky images), stored XSS, peer-to-peer ReDoS. To that extent, future work should develop an end-to-end taint analysis that allows taint propagation between client, server and the database. In this way, the security analyst can write comprehensive policies that cover realistic system-level interactions.

WEBASSEMBLY PROGRAM ANALYSIS WebAssembly is a novel binary format for the web that is supported by all the major browsers and server-side platforms. JavaScript and WebAssembly code can easily communicate bidirectionally, therefore, future work on program analysis of web applications should consider information flows across this language boundary. Fortunately, WebAssembly was designed with program verification in mind [Haa+17] and first analysis frameworks are starting to appear [LP19].

COMPARISON OF ALGORITHMIC COMPLEXITY ATTACKS Considering our work on denial-of-service attacks in Chapter 5, a related research idea is the comparison of algorithmic complexity attacks on different

web server architectures. More precisely, one should compare single-threaded, event-driven environments, e.g., Node.js, with more traditional web servers, e.g., Apache or Tomcat, from an attack bandwith perspective.

Reusable Vulnerabilities and Exploits Another idea for future work is to aggregate a benchmark suite of vulnerabilities in JavaScript code, together with their exploits. Such a suite can be used for comparing existing and future solutions aimed at hardening the security of JavaScript code. Moreover, it can serve as an example of bad practices for developers to avoid. An initial proof for the claim that such benchmarks are needed in the community is that the exploits we wrote for evaluating Synode were reused by multiple other research groups to evaluate their tools.

Information Gathering from Code or System Behavior Our results in Chapter 3 and Chapter 5 provide early evidence that an attacker can obtain important information about the components and tools used on the server-side by analyzing either the response of the server to certain requests or the structure of the client-side code. Future work should pursue this idea further by analyzing the correlation between libraries and idioms used on the client-side and the ones on the server-side. Under the assumption that full-stack developers easily migrate between the two, with high probability such correlations should exist. Another research hypothesis worth exploring is whether redundant regular expression checks deployed both on the client- and on the server-side can be used to improve our detection of websites vulnerable to ReDoS attacks.

BIBLIOGRAPHY

[Abd+17] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. "Why Do Developers Use Trivial Packages? An Empirical Case Study on npm". In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2017.

[Aca+13] Gunes Acar, Marc Juárez, Nick Nikiforakis, Claudia Díaz, Seda F. Gürses, Frank Piessens, and Bart Preneel. "FPDetective: dusting the web for fingerprinters". In: *Conference on Computer and Communications Security (CCS)*. 2013.

[Aca+14] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juárez, Arvind Narayanan, and Claudia Díaz. "The Web Never Forgets: Persistent Tracking Mechanisms in the Wild". In: *Conference on Computer and Communications Security (CCS)*. 2014.

[AHS17] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. "Measuring login webpage security". In: *Symposium on Applied Computing (SAC)*. 2017.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley, 1986.

[AT+15] Ismail Adel AL-Taharwa, Hahn-Ming Lee, Albert B. Jeng, Kuo-Ping Wu, Cheng-Seen Ho, and Shyi-Ming Chen. "JSOD: JavaScript Obfuscation Detector". In: 8 (2015).

[AMP16] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. "Understanding asynchronous interactions in full-stack JavaScript". In: *International Conference on Software Engineering (ICSE)*. 2016.

[All+18] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. "A Survey of Machine Learning for Big Code and Naturalness". In: *ACM Computing Survey* 51.4 (2018).

[AM14] Esben Andreasen and Anders Møller. "Determinacy in static analysis for jQuery". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2014.

[And+17] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. "A Survey of Dynamic Analysis and Test Generation for JavaScript". In: *ACM Computing Survey* (2017).

[AB16] Steven Arzt and Eric Bodden. "StubDroid: automatic inference of precise data-flow summaries for the android framework". In: *International Conference on Software Engineering (ICSE)*. 2016.

[Arz+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps". In: *Programming Language Design and Implementation (PLDI)*. 2014.

[AF09] Thomas H. Austin and Cormac Flanagan. "Efficient purely-dynamic information flow analysis". In: *Workshop on Programming Languages and Analysis for Security (PLAS)*. 2009.

[AF12] Thomas H. Austin and Cormac Flanagan. "Multiple facets for dynamic information flow". In: *Symposium on Principles of Programming Languages (POPL)*. 2012.

[AF10] Thomas H. Austin and Cormac Flanagan. "Permissive dynamic information flow analysis". In: *Workshop on Programming Languages and Analysis for Security (PLAS)*. 2010.

[Bab] *Babel JavaScript compiler*. `https://babeljs.io`. Accessed: 2018-02-08.

[BBD16] Michael Backes, Sven Bugiel, and Erik Derr. "Reliable Third-Party Library Detection in Android and its Security Applications". In: *Conference on Computer and Communications Security (CCS)*. 2016.

[BI16] Arturs Backurs and Piotr Indyk. "Which Regular Expression Patterns Are Hard to Match?" In: *Symposium on Foundations of Computer Science (FOCS)*. 2016.

[Bad+19] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. "Getafix: learning to fix bugs automatically". In: (2019).

[BSS17] Musard Balliu, Daniel Schoepe, and Andrei Sabelfeld. "We Are Family: Relating Information-Flow Trackers". In: *European Symposium on Research in Computer Security (ESORICS)*. 2017.

[Bao+10] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. "Strict control dependence and its effect on dynamic information flow analyses". In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2010.

[BJM08] Adam Barth, Collin Jackson, and John C. Mitchell. "Robust defenses for cross-site request forgery". In: *Conference on Computer and Communications Security (CCS)*. 2008.

[Bar+14] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Eric Medvet, and Enrico Sorio. "Automatic Synthesis of Regular Expressions from Examples". In: *IEEE Computer* 47 (2014).

[Bar+16] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. "Can a Machine Replace Humans in Building Regular Expressions? A Case Study". In: *IEEE Intelligent Systems* (2016).

[Bas+18] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. "Active learning of points-to specifications". In: *Programming Language Design and Implementation (PLDI)*. 2018.

[BBJ10] Daniel Bates, Adam Barth, and Collin Jackson. "Regular expressions considered harmful in client-side XSS filters". In: *International Conference on World Wide Web (WWW)*. 2010.

[Bau+15] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. "Run-time Monitoring and Formal Analysis of Information Flows in Chromium". In: *Network and Distributed System Security Symposium (NDSS)*. 2015.

[BPS18] Rohan Bavishi, Michael Pradel, and Koushik Sen. "Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts". In: *CoRR* arXiv:1809.05193 (2018).

[Bav+13] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. "The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache". In: *International Conference on Software Maintenance (ICSM)*. 2013.

[BDM14] Martin Berglund, Frank Drewes, and Brink van der Merwe. "Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching". In: *International Conference on Automata and Formal Languages (AFL)*. 2014.

[Bic+14] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. "Information Flow Control in WebKit's JavaScript Bytecode". In: *International Conference on Principles of Security and Trust (POST)*. 2014.

[Bic+17] Abhishek Bichhawat, Vineet Rajani, Jinank Jain, Deepak Garg, and Christian Hammer. "WebPol: Fine-Grained Information Flow Policies for Web Browsers". In: *European Symposium on Research in Computer Security (ESORICS)*. 2017.

[BHS12] Arnar Birgisson, Daniel Hedin, and Andrei Sabelfeld. "Boosting the Permissiveness of Dynamic Information-Flow Tracking by Testing". In: *European Symposium on Research in Computer Security (ESORICS)*. 2012.

[BV08] Prithvi Bisht and V. N. Venkatakrishnan. "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2008.

[Blo+14a] Remco Bloemen, Chintan Amrit, Stefan Kuhlmann, and Gonzalo Ordóñez-Matamoros. "Gentoo package dependencies over time". In: *Working Conference on Mining Software Repositories (MSR)*. 2014.

[Blo+14b] Stevens Le Blond, Adina Uritesc, Cédric Gilbert, Zheng Leong Chua, Prateek Saxena, and Engin Kirda. "A Look at Targeted Attacks Through the Lense of an NGO". In: *USENIX Security Symposium*. 2014.

[Bog+16] Christopher Bogart, Christian Kästner, James D. Herbsleb, and Ferdian Thung. "How to break an API: cost negotiation and community values in three software ecosystems". In: *International Symposium on Foundations of Software Engineering (FSE)*. 2016.

[Bro+17] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson R. Engler, Ranjit Jhala, and Deian Stefan. "Finding and Preventing Bugs in JavaScript Bindings". In: *Symposium on Security and Privacy (S&P)*. 2017.

[BWS05] Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. "Using parse tree validation to prevent SQL injection attacks". In: *Workshop on Software Engineering and Middleware (SEM)*. 2005.

[BJS09] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. "WISE: Automated test generation for worst-case complexity". In: *International Conference on Software Engineering (ICSE)*. 2009.

[Cah+16] Aaron Cahn, Scott Alfeld, Paul Barford, and S. Muthukrishnan. "An Empirical Study of Web Cookies". In: *International Conference on World Wide Web (WWW)*. 2016.

[Cal+17] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. "Surviving the Web: A Journey into Web Session Security". In: *ACM Computing Survey* 50 (2017).

[CLW17] Yinzhi Cao, Song Li, and Erik Wijmans. "(Cross-)Browser Fingerprinting via OS and Hardware Level Features". In: *Network and Distributed System Security Symposium (NDSS)*. 2017.

[Cec+15] Mariano Ceccato, Andrea Capiluppi, Paolo Falcarin, and Cornelia Boldyreff. "A large study on the effect of code obfuscation on the quality of Java code". In: *Empirical Software Engineering* 20 (2015).

[Cec+16] Mariano Ceccato, Paolo Falcarin, Alessandro Cabutto, Yosief Weldezghi Frezghi, and Cristian-Alexandru Staicu. "Search Based Clustering for Protecting Software with Diversified Updates". In: *International Symposium on Search Based Software Engineering (SSBSE)*. 2016.

[CF07] Deepak Chandra and Michael Franz. "Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine". In: *Annual Computer Security Applications Conference (ACSAC)*. 2007.

[Cha+09] Richard M. Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. "Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities". In: *Computer Security Foundations Symposium (CSF)*. 2009, 186.

[CS16a] Carl Chapman and Kathryn T. Stolee. "Exploring regular expression usage and context in Python". In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2016.

[Cha+18] Kyriakos C. Chatzidimitriou, Michail D. Papamichail, Themistoklis G. Diamantopoulos, Michail Tsapanos, and Andreas L. Symeonidis. "npm-miner: an infrastructure for measuring the quality of the npm registry". In: *International Conference on Mining Software Repositories (MSR)*. 2018.

[CS16b] Ming Cheung and James She. “Evaluating the privacy risk of user-shared images”. In: *Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 12 (2016).

[Chi+19] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin T. Vechev. “Scalable taint specification inference with big code”. In: *Programming Language Design and Implementation (PLDI)*. 2019.

[CN15] Andrey Chudnov and David A. Naumann. “Inlined Information Flow Monitoring for JavaScript”. In: *Conference on Computer and Communications Security (CCS)*. 2015.

[Chu+09] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. “Staged information flow for JavaScript”. In: *Programming Language Design and Implementation (PLDI)*. 2009.

[CAA15] Lazaro Clapp, Saswat Anand, and Alex Aiken. “Modelgen: mining explicit information flow specifications from concrete executions”. In: *International Symposium on Software Testing and Analysis (ISSTA)*. Ed. by Michal Young and Tao Xie. 2015.

[CLO07] James A. Clause, Wanchun Li, and Alessandro Orso. “Dytan: a generic dynamic taint analysis framework”. In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2007.

[CM17] Eleni Constantinou and Tom Mens. “An empirical comparison of developer retention in the RubyGems and npm software ecosystems”. In: *Innovations in Systems and Software Engineering (ISSE)* (2017).

[CW03] Scott A. Crosby and Dan S. Wallach. “Denial of Service via Algorithmic Complexity Attacks”. In: *USENIX Security Symposium*. 2003.

[Cur+11] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. “ZOZZLE: Fast and Precise In-browser JavaScript Malware Detection”. In: *USENIX Security Symposium*. 2011.

[CM10] Tom Van Cutsem and Mark S. Miller. “Proxies: design principles for robust object-oriented intercession APIs”. In: *Symposium on Dynamic Languages (DLS)*. 2010.

[DKL17] James Davis, Gregor Kildow, and Dongyoon Lee. "The Case of the Poisoned Event Handler: Weaknesses in the Node.js Event-Driven Architecture". In: *European Workshop on Systems Security EUROSEC*. 2017.

[DTL17] James Davis, Arun Thekumparampil, and Dongyoon Lee. "Node.fz: Fuzzing the Server-Side Event-Driven Architecture". In: *European Conference on Computer Systems (EuroSys)*. 2017.

[DWL18] James C. Davis, Eric R. Williamson, and Dongyoon Lee. "A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning". In: *USENIX Security Symposium*. 2018.

[Dav+18] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. "The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale". In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2018.

[DG+12] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. "FlowFox: A Web Browser with Flexible and Precise Information Flow Control". In: *Conference on Computer and Communications Security (CCS)*. 2012.

[DMC17] Alexandre Decan, Tom Mens, and Maëlick Claes. "An empirical comparison of dependency issues in OSS packaging ecosystems". In: *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017.

[DMC16] Alexandre Decan, Tom Mens, and Maëlick Claes. "On the topology of package dependency networks: a comparison of three programming language ecosystems". In: *European Conference on Software Architecture Workshops (ECSAW)*. 2016, 21.

[DMC18] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the Evolution of Technical Lag in the npm Package Dependency Network". In: *International Conference on Software Maintenance and Evolution (ICSME)*. 2018.

[DMG19] Alexandre Decan, Tom Mens, and Philippe Grosjean. "An empirical comparison of dependency network evolution in seven software packaging ecosystems". In: *Empirical Software Engineering* (2019).

[Den82] Dorothy Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[Den76] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". In: *Communications of the ACM* 19 (1976).

[DD77] Dorothy E. Denning and Peter J. Denning. "Certification of programs for secure information flow". In: *Communications of the ACM* (1977).

[DG09] Mohan Dhawan and Vinod Ganapathy. "Analyzing Information Flow in JavaScript-Based Browser Extensions". In: *Annual Computer Security Applications Conference (ACSAC)*. 2009.

[Die+17] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. "Evil Pickles: DoS Attacks Based on Object-Graph Engineering". In: *European Conference on Object-Oriented Programming (ECOOP)*. 2017.

[Dou+11] Adam Doupé, Bryce Boe, Christopher Kruegel, and Giovanni Vigna. "Fear the EAR: discovering and mitigating execution after redirect vulnerabilities". In: *Conference on Computer and Communications Security (CCS)*. 2011.

[Dur+14] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. "The Matter of Heartbleed". In: *Internet Measurement Conference (IMC)*. 2014.

[Eck10] Peter Eckersley. "How Unique Is Your Web Browser?" In: *International Symposium on Privacy Enhancing Technologies (PETS)*. 2010.

[EM20] André Takeshi Endo and Anders Møller. "NodeRacer: Event Race Detection for Node. js Applications". In: (2020).

[EN16] Steven Englehardt and Arvind Narayanan. "Online Tracking: A 1-million-site Measurement and Analysis". In: *Conference on Computer and Communications Security (CCS)*. 2016.

[Eng+15] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan Mayer, Arvind Narayanan, and Edward W. Felten. "Cookies That Give You Away: The Surveillance Implications of Web Tracking". In: *International Conference on World Wide Web (WWW)*. 2015.

[Fas+18] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. “JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2018, 303.

[Fel+13] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. “Efficient construction of approximate call graphs for JavaScript IDE services”. In: *International Conference on Software Engineering (ICSE)*. 2013.

[Fen74] Jeffrey S. Fento. “Memoryless Subsystems”. In: *The Computer Journal* 17 (1974).

[FAW13] Matthew Finifter, Devdatta Akhawe, and David A. Wagner. “An Empirical Study of Vulnerability Rewards Programs”. In: *USENIX Security Symposium*. 2013.

[GLR09] Vijay Ganesh, Tim Leek, and Martin C. Rinard. “Taint-based directed whitebox fuzzing”. In: *International Conference on Software Engineering (ICSE)*. 2009.

[GHJ18] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. “AFFOGATO: runtime detection of injection attacks for Node.js”. In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2018.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically rigorous Java performance evaluation”. In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2007.

[GAH13] Daniel M. Germán, Bram Adams, and Ahmed E. Hassan. “The Evolution of the R Software Ecosystem”. In: *European Conference on Software Maintenance and Reengineering (CSMR)*. 2013.

[GJN15] Tom van Goethem, Wouter Joosen, and Nick Nikiforakis. “The Clock is Still Ticking: Timing Attacks in the Modern Web”. In: *Conference on Computer and Communications Security (CCS)*. 2015.

[GM82] J. A. Goguen and J. Meseguer. “Security Policies and Security Models”. In: *Symposium on Security and Privacy (S&P)*. 1982.

[Gon18] Liang Gong. “Dynamic Analysis for JavaScript Code”. PhD thesis. University of California, Berkeley, 2018.

[GPS15] Liang Gong, Michael Pradel, and Koushik Sen. "JITProf: Pinpointing JIT-unfriendly JavaScript Code". In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2015.

[Gon+15] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. "DLint: Dynamically Checking Bad Coding Practices in JavaScript". In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2015.

[GMP14] Willem De Groef, Fabio Massacci, and Frank Piessens. "NodeSentry: least-privilege library integration for server-side JavaScript". In: *Annual Computer Security Applications Conference (ACSAC)*. 2014.

[GL09] Salvatore Guarnieri and Benjamin Livshits. "GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code". In: *USENIX Security Symposium*. 2009.

[Gua+11] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. "Saving the world wide web from vulnerable JavaScript". In: *International Symposium on Software Testing and Analysis (ISSTA)*. Ed. by Matthew B. Dwyer and Frank Tip. 2011.

[Haa+17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. "Bringing the web up to speed with WebAssembly". In: *Programming Language Design and Implementation (PLDI)*. 2017.

[HP18] Andrew Habib and Michael Pradel. "Is this class thread-safe? inferring documentation using graph-based learning". In: *International Conference on Automated Software Engineering (ASE)*. 2018.

[HP19] Andrew Habib and Michael Pradel. "Neural Bug Finding: A Study of Opportunities and Challenges". In: *CoRR* abs/1906.00307 (2019).

[HGM17] Mahmoud Hammad, Joshua Garcia, and Sam Malek. "A Large-Scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-Malware Products". In: *International Conference on Software Engineering (ICSE)*. 2017.

[Har+14] Seth Hardy, Masashi Crete-Nishihata, Katharine Kleemola, Adam Senft, Byron Sonne, Greg Wiseman, Phillipa Gill, and Ronald J. Deibert. "Targeted Threat Index: Characterizing and Quantifying Politically-Motivated Targeted Malware". In: *USENIX Security Symposium*. 2014.

[HS12] Daniel Hedin and Andrei Sabelfeld. "Information-Flow Security for a Core of JavaScript". In: *Computer Security Foundations Symposium (CSF)*. 2012.

[Hed+17] Daniel Hedin, Alexander Sjösten, Frank Piessens, and Andrei Sabelfeld. "A Principled Approach to Tracking Information Flow in the Presence of Libraries". In: *International Conference on Principles of Security and Trust (POST)*. 2017.

[Hed+14] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. "JSFlow: tracking information flow in JavaScript and its APIs". In: *Symposium on Applied Computing (SAC)*. 2014.

[Hei+11] Mario Heiderich, Tilman Frosch, Meiko Jensen, and Thorsten Holz. "Crouching tiger - hidden payload: security risks of scalable vectors graphics". In: *Conference on Computer and Communications Security (CCS)*. 2011.

[Hei+14] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. "Scriptless attacks: Stealing more pie without touching the sill". In: *Journal of Computer Security* 22 (2014).

[Hel+18] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. "Deep learning type inference". In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2018.

[HSC15] Stefan Heule, Manu Sridharan, and Satish Chandra. "Mimic: computing models for opaque code". In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2015.

[Hoo+11] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. "Fast and Precise Sanitizer Analysis with BEK". In: *USENIX Security Symposium*. 2011.

[HZ16] Matthias Höschele and Andreas Zeller. "Mining input grammars from dynamic taints". In: *International Conference on Automated Software Engineering (ASE)*. 2016.

[Hua+15] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. “From System Services Freezing to System Server Shutdown in Android: All You Need Is a Loop in an App”. In: *Conference on Computer and Communications Security (CCS)*. 2015.

[JJM12] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. “Remedying the eval that men do”. In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2012.

[JMT09] Simon Holm Jensen, Anders Møller, and Peter Thiemann. “Type Analysis for JavaScript”. In: *International Symposium on Static Analysis (SAS)*. 2009.

[Jen+15] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. “MemInsight: platform-independent memory debugging for JavaScript”. In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2015.

[Jia+16] Yaoqi Jia, Zheng Leong Chua, Hong Hu, Shuo Chen, Prateek Saxena, and Zhenkai Liang. “"The Web/Local" Boundary Is Fuzzy: A Security Study of Chrome’s Process-based Sandboxing”. In: *Conference on Computer and Communications Security (CCS)*. 2016.

[Jin+14] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. “Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation”. In: *Conference on Computer and Communications Security (CCS)*. 2014.

[JAP15] Mehran Jodavi, Mahdi Abadi, and Elham Parhizkar. “JSObfus-Detector: A binary PSO-based one-class classifier ensemble to detect obfuscated JavaScript code”. In: *International Symposium on Artificial Intelligence and Signal Processing (AISP)* (2015).

[Joh08] Martin Johns. “On JavaScript Malware and related threats”. In: *Journal in Computer Virology* 4 (2008).

[KJ11] Jaap Kabbedijk and Slinger Jansen. “Steering Insight: An Exploration of the Ruby Software Ecosystem”. In: *Software Business - Second International Conference (ICSOB)*. 2011.

[Kan+11] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. “DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation”. In: *Network and Distributed System Security Symposium (NDSS)*. 2011.

[Kap+11] Scott Kaplan, Ben Livshits, Ben Zorn, Christian Siefert, and Charlie Cursinger. *"NOFUS: Automatically Detecting" + String.fromCharCode(32) + "ObFuSCateD ".toLowerCase() + "JavaScript Code"*. Tech. rep. 2011.

[Kar+18] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. "Platform-independent dynamic taint analysis for javascript". In: *IEEE Transactions on Software Engineering* (2018).

[KT15] Matthias Keil and Peter Thiemann. "TreatJS: Higher-Order Contracts for JavaScripts". In: *European Conference on Object-Oriented Programming (ECOOP)*. 2015.

[Ker+13] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. "CrowdFlow: Efficient Information Flow Security". In: *International Conference on Information Security (ISC)*. 2013.

[Kie+09] Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. "Automatic creation of SQL Injection and cross-site scripting attacks". In: *International Conference on Software Engineering (ICSE)*. 2009.

[Kik+17] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. "Structure and evolution of package dependency networks". In: *International Conference on Mining Software Repositories (MSR)*. 2017.

[Kin+08] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. "Implicit Flows: Can't Live with 'Em, Can't Live without 'Em". In: *International Conference on Information Systems Security (ICISS)*. 2008.

[KRT13] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. "Static Analysis for Regular Expression Denial-of-Service Attacks". In: *International Conference on Network and System Security (NSS)*. 2013.

[Koc+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *Symposium on Security and Privacy (S&P)*. 2019.

[Kot+13] Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. "Cross-origin pixel stealing: timing attacks using CSS filters". In: *Conference on Computer and Communications Security (CCS)*. 2013.

[KM19] Erik Krogh Kristensen and Anders Møller. "Reasonably-most-general clients for JavaScript library analysis". In: *International Conference on Software Engineering (ICSE)*. 2019.

[Kul+17] Raula Gaikovina Kula, Ali Ouni, Daniel M. Germán, and Katsuro Inoue. "On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem". In: *CoRR* abs/1709.04638 (2017).

[LRB16] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. "Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints". In: *Symposium on Security and Privacy (S&P)*. 2016.

[LK16] Eric Larson and Anna Kirk. "Generating Evil Test Strings for Regular Expressions". In: *International Conference on Software Testing, Verification and Validation (ICST)*. 2016.

[Lau+17] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. "Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web". In: *Network and Distributed System Security Symposium (NDSS)*. 2017.

[LP19] Daniel Lehmann and Michael Pradel. "Wasabi: A Framework for Dynamically Analyzing WebAssembly". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[LSJ13] Sebastian Lekies, Ben Stock, and Martin Johns. "25 million flows later: large-scale detection of DOM-based XSS". In: *Conference on Computer and Communications Security (CCS)*. 2013.

[Lek+15] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. "The Unexpected Dangers of Dynamic JavaScript". In: *USENIX Security Symposium*. 2015.

[Ler+16] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. "Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016". In: *USENIX Security Symposium*. 2016.

[Ler+17] Nuttapon Lertwittayatrai, Raula Gaikovina Kula, Saya Onoue, Hideaki Hata, Arnon Rungsawang, Pattara Leelaprute, and Kenichi Matsumoto. "Extracting Insights from the Topology of the JavaScript Package Ecosystem". In: *Asia-Pacific Software Engineering Conference (APSEC)*. 2017.

[Li+17] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. "Libd: Scalable and precise third-party library detection in Android markets". In: *International Conference on Software Engineering (ICSE)*. 2017.

[Li+09] Nuo Li, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. "Reggae: Automated Test Generation for Programs Using Complex Regular Expressions". In: *International Conference on Automated Software Engineering (ASE)*. 2009.

[Lib15] Timothy Libert. "Exposing the Hidden Web: An Analysis of Third-Party HTTP Requests on 1 Million Websites". In: *International Journal of Communication* 9 (2015).

[LJJ09] Peter Likarish, Eunjin Jung, and Insoon Jo. "Obfuscated malicious JavaScript detection using classification techniques". In: *International Conference on Malicious and Unwanted Software (MALWARE)*. 2009.

[Liu+16] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. "Nomen est omen: exploring and exploiting similarities between argument and parameter names". In: *International Conference on Software Engineering (ICSE)*. 2016.

[LC13] Benjamin Livshits and Stephen Chong. "Towards fully automatic placement of security sanitizers and declassifiers". In: *Symposium on Principles of Programming Languages (POPL)*. 2013.

[Liv+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. "In defense of soundiness: a manifesto". In: *Communications of the ACM* 58 (2015).

[LV09] Mike Ter Louw and V. N. Venkatakrishnan. "Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers". In: *Symposium on Security and Privacy (S&P)*. 2009.

[LD12] Gen Lu and Saumya K. Debray. "Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach". In: *International Conference on Software Security and Reliability (SERE)*. 2012.

[MTK12] Alex Mackey, William Stewart Tulloch, and Mahesh Krishnan. *Introducing. NET 4.5*. Apress, 2012, 49.

[MTL15a] Magnus Madsen, Frank Tip, and Ondrej Lhoták. "Static analysis of event-driven Node.js JavaScript applications". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2015.

[Mad+16] Magnus Madsen, Frank Tip, Esben Andreasen, Koushik Sen, and Anders Møller. "Feedback-directed instrumentation for deployed JavaScript applications". In: *International Conference on Software Engineering (ICSE)*. 2016.

[MPP19] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. "NL2Type: inferring JavaScript function types from natural language information". In: *International Conference on Software Engineering (ICSE)*. 2019.

[Man16] Konstantinos Manikas. "Revisiting software ecosystems Research: A longitudinal literature study". In: *Journal of Systems and Software* 117 (2016).

[Mar+14] William R. Marczak, John Scott-Railton, Morgan Marquis-Boire, and Vern Paxson. "When Governments Hack Opponents: A Look at Actors and Technology". In: *USENIX Security Symposium*. 2014.

[MP09] Wes Masri and Andy Podgurski. "Measuring the strength of information flows in programs". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 19 (2009).

[Mas+15] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. "Use at your own risk: the Java unsafe API in the wild". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2015.

[MM12] Jonathan R. Mayer and John C. Mitchell. "Third-Party Web Tracking: Policy and Technology". In: *Symposium on Security and Privacy (S&P)*. 2012.

[Mea+12] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. “Eval begone!: semi-automated removal of eval from JavaScript programs”. In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2012.

[Mel+18] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. “Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting”. In: *Network and Distributed System Security Symposium (NDSS)*. 2018.

[Men16] Tom Mens. “An Ecosystemic and Socio-Technical View on Software Maintenance and Evolution”. In: *International Conference on Software Maintenance and Evolution (ICSME)*. 2016.

[MMT18] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. “Type Regression Testing to Detect Breaking Changes in Node.js Libraries”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2018.

[Mil06] Mark Miller. “Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control”. PhD thesis. Johns Hopkins University, 2006.

[Mit+16] Dimitris Mitropoulos, Konstantinos Stroggylos, Diomidis Spinellis, and Angelos D Keromytis. “How to Train Your Browser: Preventing XSS Attacks Using Contextual Script Fingerprints”. In: *ACM Transactions on Privacy and Security (TOPS)* 19 (2016).

[MT19] Anders Møller and Martin Toldam Torp. “Model-based testing of breaking changes in Node.js libraries”. In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2019.

[Mou+16] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. “Convolutional Neural Networks over Tree Structures for Programming Language Processing”. In: *AAAI Conference on Artificial Intelligence*. 2016.

[MS12] Keaton Mowery and Hovav Shacham. “Pixel perfect: Fingerprinting canvas in HTML5”. In: *Web 2.0 Security & Privacy, (W2SP)*. 2012.

[MTL15b] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. "Detecting JavaScript Races that Matter". In: *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. 2015.

[ML00] Andrew C. Myers and Barbara Liskov. "Protecting privacy using the decentralized label model". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9 (2000).

[NS05] James Newsome and Dawn Xiaodong Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature-Generation of Exploits on Commodity Software". In: *Network and Distributed System Security Symposium (NDSS)*. 2005.

[NHG19] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. "Nodest: feedback-driven static analysis of Node.js applications". In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2019.

[NNH05] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005.

[Nik+13] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank PieWssens, and Giovanni Vigna. "Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting". In: *Symposium on Security and Privacy (S&P)*. 2013.

[Nik+12] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. "You are what you include: large-scale evaluation of remote JavaScript inclusions". In: *Conference on Computer and Communications Security (CCS)*. 2012.

[OD12] Andres Ojamaa and Karl Düüna. "Assessing the security of Node.js platform". In: *International Conference for Internet Technology and Secured Transactions (ICITST)*. 2012.

[Pan+16] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. "CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites". In: *Conference on Computer and Communications Security (CCS)*. 2016.

[PJR19] Joonyoung Park, Alexander Jordan, and Sukyoung Ryu. "Automatic Modeling of Opaque Code for JavaScript Static Analysis". In: *Fundamental Approaches to Software Engineering (FASE)*. 2019.

[PDP18] Jibesh Patra, Pooja N. Dixit, and Michael Pradel. "ConflictJS: Finding and Understanding Conflicts Between JavaScript Libraries". In: *International Conference on Software Engineering (ICSE)*. 2018.

[Pel+17] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. "Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs". In: *Conference on Computer and Communications Security (CCS)*. 2017.

[Pet+17] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities". In: *Conference on Computer and Communications Security (CCS)*. 2017.

[PO17] Brian Pfretzschner and Lotfi Ben Othmane. "Identification of Dependency-based Attacks on Node.js". In: *International Conference on Availability, Reliability and Security (ARES)*. 2017.

[PHG14] Michael Pradel, Markus Huggler, and Thomas R. Gross. "Performance Regression Testing of Concurrent Classes". In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2014.

[PSS15] Michael Pradel, Parker Schuh, and Koushik Sen. "TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript". In: *International Conference on Software Engineering (ICSE)*. 2015.

[PS18] Michael Pradel and Koushik Sen. "DeepBugs: a learning approach to name-based bug detection". In: (2018).

[Pra+14] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. "EventBreak: Analyzing the Responsiveness of User Interfaces through Performance-Guided Test Generation". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2014.

[RJLS10] Dongseok Jang andZhaoGL15 Ranjit Jhala, Sorin Lerner, and Hovav Shacham. “An empirical study of privacy-violating information flows in JavaScript web applications”. In: *Conference on Computer and Communications Security (CCS)*. 2010.

[RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. “A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks.” In: *Network and Distributed System Security Symposium (NDSS)*. 2014.

[RL12] Donald Ray and Jay Ligatti. “Defining code-injection attacks”. In: *Symposium on Principles of Programming Languages (POPL)*. 2012.

[RVK15] Veselin Raychev, Martin T. Vechev, and Andreas Krause. “Predicting Program Properties from "Big Code".” In: *Symposium on Principles of Programming Languages (POPL)*. 2015.

[Ray+16] Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. “Learning programs from noisy data”. In: *Symposium on Principles of Programming Languages (POPL)*. 2016.

[Rei+16] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. “Call graph construction for Java libraries”. In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2016.

[Ric+11] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. “The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2011.

[RV18] Diego Rodríguez-Baquero and Mario Linares Vásquez. “Mutode: generic JavaScript and Node.js mutation testing tool”. In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2018.

[RKW12] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. “Detecting and Defending Against Third-Party Tracking on the Web”. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2012.

[RSL09] A. Russo, A. Sabelfeld, and K. Li. “Implicit flows in malicious and nonmalicious code”. In: *Marktoberdorf Summer School (IOS Press)* (2009).

[SM03] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security". In: *IEEE Journal on Selected Areas in Communications* 21 (2003).

[SSS11] Mike Samuel, Prateek Saxena, and Dawn Song. "Context-sensitive auto-sanitization in web templating languages using type qualifiers". In: *Conference on Computer and Communications Security (CCS)*. 2011.

[SML11] Prateek Saxena, David Molnar, and Benjamin Livshits. "SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications". In: *Conference on Computer and Communications Security (CCS)*. 2011.

[Sax+10] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. "FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications". In: *Network and Distributed System Security Symposium (NDSS)*. 2010.

[SKA17] Mohammed Sayagh, Noureddine Kerzazi, and Bram Adams. "On cross-stack configuration errors". In: *International Conference on Software Engineering (ICSE)*. 2017.

[Sch+13] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. "Dynamic determinacy analysis". In: *Programming Language Design and Implementation (PLDI)*. 2013.

[Sch+16] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. "Explicit Secrecy: A Policy for Taint Tracking". In: *European Symposium on Security and Privacy (EuroS&P)*. 2016.

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)". In: *Symposium on Security and Privacy (S&P)*. 2010.

[SP16] Marija Selakovic and Michael Pradel. "Performance Issues and Optimizations in JavaScript: An Empirical Study". In: *International Conference on Software Engineering (ICSE)*. 2016.

[Sel+18] Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. "Test generation for higher-order functions in dynamic languages". In: *PACMPL* 2.OOPSLA (2018).

[Sen+13] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. "Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of JavaScript". In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2013.

[SM15] Alexander Serebrenik and Tom Mens. "Challenges in Software Ecosystems Research". In: *European Conference on Software Architecture Workshops (ECSAW)*. 2015.

[SMS12] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. "Automated Discovery of Valid Test Strings from the Web Using Dynamic Regular Expressions Collation and Natural Language Processing". In: *International Conference on Quality Software (QSIC)*. 2012.

[SWP17] Huasong Shan, Qingyang Wang, and Calton Pu. "Tail Attacks on Web Applications". In: *Conference on Computer and Communications Security (CCS)*. 2017.

[Shi] Chris Shiflett. *Cross-Site Request Forgeries*. http://shiflett.org/articles/cross-site-request-forgeries.

[SPK16] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. "The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information". In: *Symposium on Security and Privacy (S&P)*. 2016.

[SAS17] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. "Discovering Browser Extensions via Web Accessible Resources". In: 2017.

[SSP19] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. "Anything to Hide? Studying Minified and Obfuscated Code in the Web". In: *The Web Conference (WWW)*. 2019.

[STK17] Peter Snyder, Cynthia Taylor, and Chris Kanich. "Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security". In: *Conference on Computer and Communications Security (CCS)*. 2017.

[SS13] Sooel Son and Vitaly Shmatikov. "The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites". In: *Network and Distributed System Security Symposium (NDSS)*. 2013.

[SBS15] Pratik Soni, Enrico Budianto, and Prateek Saxena. "The SICILIAN Defense: Signature-based Whitelisting of Web JavaScript". In: *Conference on Computer and Communications Security (CCS)*. 2015.

[SE13] Aditya K. Sood and Richard J. Enbody. "Targeted Cyberattacks: A Superset of Advanced Persistent Threats". In: *IEEE Security & Privacy* 11 (2013).

[SL19] Thodoris Sotiropoulos and Benjamin Livshits. "Static Analysis for Asynchronous JavaScript Programs". In: *European Conference on Object-Oriented Programming (ECOOP)*. 2019.

[SP18] Cristian-Alexandru Staicu and Michael Pradel. "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers". In: *USENIX Security Symposium*. 2018.

[SP19] Cristian-Alexandru Staicu and Michael Pradel. "Leaky Images: Targeted Privacy Attacks in the Web". In: *USENIX Security Symposium*. 2019.

[SPL16] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. *Understanding and Automatically Preventing Injection Attacks on Node.js*. Tech. rep. TUD-CS-2016-14663. TU Darmstadt, Department of Computer Science, 2016.

[SPL18] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. "SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS". In: *Network and Distributed System Security Symposium (NDSS)*. 2018.

[Sta+19] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. "An Empirical Study of Information Flows in Real-World JavaScript". In: *Workshop on Programming Languages and Analysis for Security (PLAS)*. 2019.

[Sta+20] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. "Extracting Taint Specifications for JavaScript Libraries". In: *International Conference on Software Engineering (ICSE)*. 2020.

[Sto+18] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. "Didn't You Hear Me? - Towards More Successful Web Vulnerability Notifications". In: *Network and Distributed System Security Symposium (NDSS)*. 2018.

[Sto+16] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. “Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification”. In: *USENIX Security Symposium*. 2016.

[Sto+14] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. “Precise Client-side Protection against DOM-based Cross-Site Scripting”. In: *USENIX Security Symposium*. 2014.

[Su+17] Jessica Su, Ansh Shukla, Sharad Goel, and Arvind Narayanan. “De-anonymizing Web Browsing Data with Social Networks”. In: *International Conference on World Wide Web (WWW)*. 2017.

[SW06] Zhendong Su and Gary Wassermann. “The essence of command injection attacks in web applications”. In: *Symposium on Principles of Programming Languages (POPL)*. 2006.

[Sul11] Bryan Sullivan. “Server-side JavaScript injection”. In: *Black Hat USA* (2011).

[Sun+18] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. “Efficient dynamic analysis for Node.js”. In: *International Conference on Compiler Construction (CC)*. 2018.

[TPR16] Bernhard Tellenbach, Sergio Paganoni, and Marc Rennhard. “ Detecting Obfuscated JavaScripts using Machine Learning ”. In: *International Journal on Advances in Security* 9 (2016).

[Tho68] Ken Thompson. “Programming techniques: Regular expression search algorithm”. In: *Communications of the ACM* 11 (1968).

[TPG18] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. “Synthesizing Programs that Expose Performance Bottlenecks”. In: *International Symposium on Code Generation and Optimization (CGO)*. 2018.

[Tra+12] Minh Tran, Xinshu Dong, Zhenkai Liang, and Xuxian Jiang. “Tracking the Trackers: Fast and Scalable Dynamic Analysis of Web Content for Privacy Violations”. In: *International Conference ob Applied Cryptography and Network Security (ACNS)*. 2012.

[TFP14] Omer Tripp, Pietro Ferrara, and Marco Pistoia. “Hybrid security analysis of web JavaScript code via dynamic partial evaluation”. In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2014.

[Tri+09] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. “TAJ: effective taint analysis of web applications”. In: *Programming Language Design and Implementation (PLDI)*. 2009.

[Tsc16] Nikolai Philipp Tschacher. “Typosquatting in programming language package managers”. Bachelor’s Thesis. Universität Hamburg, Fachbereich Informatik, 2016.

[Vas+18] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. “BreakApp: Automated, Flexible Application Compartmentalization”. In: *Network and Distributed System Security Symposium, (NDSS)*. 2018.

[VHT10] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. “Rex: Symbolic Regular Expression Explorer”. In: *International Conference on Software Testing, Verification and Validation (ICST)*. 2010.

[Ven+18] Giridhari Venkatadri, Athanasios Andreou, Yabing Liu, Alan Mislove, Krishna P Gummadi, Patrick Loiseau, and Oana Goga. “Privacy Risks with Facebook’s PII-based Targeting: Auditing a Data Broker’s Advertising Interface”. In: *Symposium on Security and Privacy (S&P)*. 2018.

[VPC13] Corrado Aaron Visaggio, Giuseppe Antonio Pagin, and Gerardo Canfora. “An empirical study of metric-based methods to detect obfuscated code”. In: *International Journal of Security and its Applications* (2013).

[Vog+07] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. “Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis”. In: *Network and Distributed System Security Symposium (NDSS)*. 2007.

[Wan+17] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. “A comprehensive study on real world concurrency bugs in Node.js”. In: *International Conference on Automated Software Engineering (ASE)*. 2017.

[Wan+18] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, and Tao Wei. “Software Protection on the Go: A Large-Scale Empirical Study on Mobile App Obfuscation”. In: *International Conference on Software Engineering (ICSE)*. 2018.

[WCW16] Yao Wang, Wan-dong Cai, and Peng-cheng Wei. “A deep learning approach for detecting malicious JavaScript code”. In: 9 (2016).

[WS08] Gary Wassermann and Zhendong Su. “Static detection of cross-site scripting vulnerabilities”. In: *International Conference on Software Engineering (ICSE)*. 2008.

[Was+08] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. “Dynamic test input generation for web applications”. In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2008.

[Wei15] Shiyi Wei. “Practical Analysis of the Dynamic Characteristics of JavaScript”. Virginia Tech, 2015.

[Wei+16a] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. “CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy”. In: *Conference on Computer and Communications Security (CCS)*. 2016.

[Wei+16b] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. “Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA”. In: *International Conference on Implementation and Application of Automata (CIAA)*. 2016.

[Wei+11] Zachary Weinberg, Eric Yawei Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. “I Still Know What You Visited Last Summer: Leaking Browsing History via User Interaction and Side Channel Attacks”. In: *Symposium on Security and Privacy (S&P)*. 2011.

[Wil04] Paul Wilton. *Beginning JavaScript*. John Wiley & Sons, 2004, 312.

[WSR16] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. “A look at the dynamics of the JavaScript package ecosystem”. In: *International Conference on Mining Software Repositories (MSR)*. 2016.

[Won+10] Gilbert Wondracek, Thorsten Holz, Engin Kirda, and Christopher Kruegel. “A Practical Attack to De-anonymize Social Network Users”. In: *Symposium on Security and Privacy (S&P)*. 2010.

[Wüs+17] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. “Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2017.

[XZZ13] Wei Xu, Fangfang Zhang, and Sencun Zhu. “ JStill: Mostly static detection of obfuscated malicious javascript code ”. In: *Conference on Data and Application Security and Privacy (CODASPY)* (2013).

[XZZ12] Wei Xu, Fangfang Zhang, and Sencun Zhu. “The power of obfuscation techniques in malicious JavaScript code: A measurement study”. In: *International Conference on Malicious and Unwanted Software (MALWARE)*. 2012.

[Yak+16] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. “Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study”. In: *Symposium on Security and Privacy (S&P)*. 2016.

[Yu+07] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. “JavaScript instrumentation for browser security”. In: *Symposium on Principles of Programming Languages (POPL)*. 2007.

[Yu+16] Zhonghao Yu, Sam Macbeth, Konark Modi, and Josep M. Pujol. “Tracking the Trackers”. In: *International Conference on World Wide Web (WWW)*. 2016.

[YW09] Chuan Yue and Haining Wang. “Characterizing insecure JavaScript practices on the web”. In: *International Conference on World Wide Web (WWW)*. 2009.

[Zda02] Stephan Zdancewic. “Programming Languages for Information Security”. PhD thesis. Cornell University, 2002.

[ZGL15] Mingyi Zhao, Jens Grossklags, and Peng Liu. “An Empirical Study of Web Vulnerability Discovery Ecosystems”. In: *Conference on Computer and Communications Security (CCS)*. 2015.

[ZZG13] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. "Z3-str: a Z3-based string solver for web application analysis". In: *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2013.

[Zim+19] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. "Small World with High Risks: A Study of Security Threats in the npm Ecosystem". In: *USENIX Security Symposium*. 2019.