Saarland University
Department of Computer Science

Studying JavaScript Security Through Static Analysis

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von
Aurore Fass

Saarbrücken, 2020

Tag des Kolloquiums:	26. Mai 2021
Dekan:	Prof. Thomas Schuster

Prüfungsausschuss:

Vorsitzender:	Prof. Christian Rossow
Berichterstattende:	Prof. Michael Backes
	Dr. Ben Stock
	Prof. Konrad Rieck
Akademischer Mitarbeiter:	Dr. Dolière Francis Somé

Zusammenfassung

Mit dem stetigen Wachstum des Internets wächst auch das Interesse von Angreifern. Ursprünglich sollte das Internet Menschen verbinden; gleichzeitig benutzen aber Angreifer diese Vernetzung, um Schadprogramme wirksam zu verbreiten. Insbesondere JavaScript ist zu einem beliebten Angriffsvektor geworden, da es Angreifer ermöglicht Bugs und weitere Sicherheitslücken auszunutzen, und somit die Sicherheit und Privatsphäre der Internetnutzern zu gefährden. In dieser Dissertation fokussieren wir uns auf die Erkennung solcher Bedrohungen, indem wir JavaScript Code statisch und effizient analysieren.

Zunächst beschreiben wir unsere zwei Detektoren, welche Methoden des maschinellen Lernens mit statischen Features aus Syntax, Kontroll- und Datenflüssen kombinieren zur Erkennung bösartiger JavaScript Dateien. Wir evaluieren daraufhin die Verlässlichkeit solcher statischen Systeme, indem wir bösartige JavaScript Dokumente umschreiben, damit sie die syntaktische Struktur von bestehenden gutartigen Skripten reproduzieren. Zuletzt studieren wir die Sicherheit von Browser Extensions. Zu diesem Zweck modellieren wir Extensions mit einem Graph, welcher Kontroll-, Daten-, und Nachrichtenflüsse mit Pointer Analysen kombiniert, wodurch wir externe Flüsse aus und zu kritischen Extension-Funktionen erkennen können. Insgesamt wiesen wir 184 verwundbare Chrome Extensions nach, welche die Angreifer ausnutzen könnten, um beispielsweise beliebigen Code im Browser eines Opfers auszuführen.

Abstract

As the Internet keeps on growing, so does the interest of malicious actors. While the Internet has become widespread and popular to interconnect billions of people, this interconnectivity also simplifies the spread of malicious software. Specifically, JavaScript has become a popular attack vector, as it enables to stealthily exploit bugs and further vulnerabilities to compromise the security and privacy of Internet users. In this thesis, we approach these issues by proposing several systems to statically analyze real-world JavaScript code at scale.

First, we focus on the detection of malicious JavaScript samples. To this end, we propose two learning-based pipelines, which leverage syntactic, control and data-flow based features to distinguish benign from malicious inputs. Subsequently, we evaluate the robustness of such static malicious JavaScript detectors in an adversarial setting. For this purpose, we introduce a generic camouflage attack, which consists in rewriting malicious samples to reproduce existing benign syntactic structures. Finally, we consider vulnerable browser extensions. In particular, we abstract an extension source code at a semantic level, including control, data, and message flows, and pointer analysis, to detect suspicious data flows from and toward an extension privileged context. Overall, we report on 184 Chrome extensions that attackers could exploit to, e.g., execute arbitrary code in a victim's browser.

Acknowledgments

This thesis concludes three intense but rewarding years of research at CISPA - Helmholtz Center for Information Security. Foremost, I would like to take the opportunity to thank the many people who supported and encouraged me during my time as a PhD student.

First, a big thank you to Michael Backes and Ben Stock, for their advice and guidance, for the fruitful discussions and their valuable feedback. It has been a privilege to work in an environment where I have the opportunity to decide on the topics of interest to me and their confidence to pursue in these research directions. I am grateful for their time and support, and I am honored to have had the chance to work closely with them and learn from them. While my thesis closes a chapter of our joint work, it also brings new opportunities, which I am looking forward to sharing with them.

Second, I would like to thank Konrad Rieck, for reviewing my thesis. In particular, his insightful questions and remarks during my defense paved the way for future work. Also, I am grateful for his feedback on my first research projects.

Third, special thanks go to Christian Rossow, for chairing my defense and making the remote session as warm as possible. Also, I am grateful for his feedback, support, and advice, as well as for letting me join his weekly group meetings.

Naturally, I would not have had such an amazing and fulfilling experience without the support and presence of my colleagues from CISPA and from Saarland University. So a big thank you to each one of them, more specifically:

- to Marius Steffens and Sebastian Roth, my incredible office mates, for creating such an awesome atmosphere, for all the exciting discussions and brainstorming sessions. I would also like to mention the numerous SWAGtivities and SWAGlettes, which were a perfect supplement to our office lives;
- to Francis Somé and Robert Krawczyk, my additional co-authors, for the fruitful discussions and research work. It has been a pleasure to have the opportunity to work with them, and I am looking forward to many more to come;
- to Cris Staicu, Giada Martina Stivala, Giancarlo Pellegrino, Pierre Laperdrix, Shubham Agarwal, and Soheil Khodayari, the remaining WebSec people, for the very nice (Webby) discussions and for having the chance to share research ideas with them and get their feedback;
- to Benedikt Birtel, Fabian Schwarz, Giorgi Maisuradze, Johannes Krupp, Jonas Bushart, Markus Bauer, and Michael Brengel, for welcoming me in Christian's group meetings and providing me with constructive feedback;
- to Marvin Moog and Markus Demmel, for being such motivated and enthusiastic students, always full of ideas and energy to implement or test them. It is a real pleasure to supervise them;
- to Nils Glörfeld, Dennis Salzmann, Anne Christin Deutschen, and Luc Seyler, our Hiwis, for their helpful contributions to several research projects;

– to Anne Monzel-Busch and Julia Schulz, for the incredible time we spent together, with a special reference to all rock-climbing sessions (and this 7, obviously);
– to Ben Ehrfeld and Sebastian Meurer, for their technical support, of course, as well as very nice barbecue evenings;
– to Bettina Balthasar, Julia Schwarz, Katharina Krombholz, Mazi Valizadeh, Olga Kill, and Sandra Strohbach, for their constant willingness to help me, for their time and availability when needed;
– to Carolyn Guthoff, Duc Nguyen, Michael Schilling, Oliver Schranz, Pascal Becher, Patrick Speicher, Sarah Cieslik, Sebastian Klöckner, and Tobias Ebelshäuser, for all the nice and spontaneous discussions.

Special thanks also go to my former professors and lecturers from TELECOM Nancy and Loria, who contributed to my being here today:
– to Olivier Festor, for encouraging me to apply for a PhD position at CISPA;
– to Sébastien Da Silva, for his concerns regarding my professional life and his availability when needed, be it just for a hot chocolate;
– to Isabelle Chrisment, Rémi Badonnel, and Thibault Cholez, for their unconditional support and trust, and for already teaching me the basics of research and paper writing;
– to Pierre Monnin, for the (mostly fortuitous) inspiring discussions, for his support and sage advice;
– to Jean-Yves Marion, for giving me the opportunity to give several talks in front of his research group. I am very much looking forward to the next one.

Furthermore, I gratefully acknowledge the help of the German Federal Office for Information Security (BSI), Kafeine DNC, and VirusTotal, which provided us with malicious JavaScript samples for our experiments.

In addition, I would like to thank my friends and family, for their support and encouragement, for their good mood and the incredible memories we share. I am delighted that so many of them could be (remotely) with me for my defense.

Beyond my friends and family, I am also really honored that so many people could attend my defense and share this important day with me.

A second round of thanks then goes to my colleagues and friends Anne, Cris, Giada, Julia, Marius, Sebastian, Shubham, and Soheil, who provided me with constructive feedback on preliminary versions of my thesis and/or of my defense.

I also dearly thank (again) Ben, Marius, Giada, Sebastian, Shubham, and Soheil, for the sweetest PhD hat :)

Finally, I would like to extend my thanks to all the persons who provided us with constructive feedback on our papers and research projects. I am also delighted to have had the opportunity to meet so many awesome people at numerous conferences. So, thank you all for the very nice discussions and the great time. And if you are reading this, thank you to you too.

Contents

List of Figures

List of Tables

List of Code Listings

List of Algorithms

1 Introduction

1.1 State of the Art and Motivation

In recent years, the Internet has become ubiquitous in our daily lives. In particular, the Web has grown into the most popular software platform, used by billions of people every day. It has considerably changed our ways of living, working, and communicating, moving our world to a more digital one, e.g., online banking, e-learning, or social networks. Given the popularity and widespread character of the Internet and, more specifically, the Web, they naturally attract the interest of malicious actors who try to leverage them as vectors for attacking their victims' machines. In the last few years, we have seen a rise in the number and impact of cyber-attacks, e.g., the botnet Mirai in 2016 [201], the ransomware outbreaks Petya [202] and WannaCry [203] both in 2017. The impact of such attacks is all the more exacerbated by our online world, which enables malware to rapidly infect victims everywhere, anytime. More specifically, the first step to harm a victim's machine often relies on JavaScript payloads [58, 123, 128, 198].

While JavaScript was initially invented to create sophisticated and interactive web pages, it can also be abused to perform malicious activities, such as drive-by download attacks [49]. While drive-by downloads mostly originate from exploit kits [113], another popular way to lead to these attacks includes malicious email attachments. As far as exploit kits are concerned, they first infect a victim's device during Web browsing before silently probing the machine for vulnerabilities they could exploit, e.g., targeting old browsers versions, Java, or Adobe Flash plugins. Once they find a vulnerability, they can subsequently launch the actual payload to do the real damage, e.g., ransomware outbreaks. Similarly, for malicious JavaScript attachments, upon opening, they will try to download and install the actual malware [66].

Even though exploit kits became popular and widespread in 2010 [134], they are still a serious threat at the time of writing. In particular, multiple exploit kits were reported in 2019 and 2020, e.g., Fallout, Magnitude, RIG, Spelevo, or Underminer [99, 116, 179]. They specifically target Adobe Flash Player (CVE-2018-4878 [206] and CVE-2018-15982 [205]) and Internet Explorer (CVE-2018-8174 [207] and CVE-2019-1367 [208]) vulnerabilities, from 2018 and 2019, before launching the actual attack, e.g., ransomware or banking trojans [77].

To impede the detection of such nefarious JavaScript files, malicious actors are abusing obfuscation techniques, which foil approaches directly relying on content matching, e.g., traditional anti-virus signatures, and impose additional hurdles to manual analysis. Prior work has been proposed to analyze and detect obfuscated malicious JavaScript inputs automatically. For example, Rieck et al. and Curtsinger et al. respectively introduced Cujo in 2010 [173] and Zozzle in 2011 [50]. To handle obfuscated files, both of them include a dynamic component to analyze unpacked code at run-time. In particular, Zozzle extracts node value features from the Abstract Syntax Tree (AST) of executed code, while Cujo also includes a static part, based on lexical units, to handle less obfuscated files. This way, both tools leverage execution traces, combined with lexical or syntactic features and machine learning algorithms, to detect recurrent malicious patterns. Still, the malicious JavaScript landscape has evolved since 2011 when, e.g., "relatively few identifier-renaming schemes [were] being employed by attackers" [50].

Besides malicious activities, JavaScript is also one of the core technologies of the Web platform, e.g., over 96% of all websites use JavaScript as a client-side programming language [212]. Given the large volume of JavaScript files in the wild, we cannot execute all of them anymore to check for maliciousness. In fact, dynamic analysis is costly, both in terms of equipment and run-time performance. Also, malicious JavaScript samples can have a specific behavior depending, e.g., on time or on the infrastructure where they are executed, meaning that a dynamic approach would not always detect them. Thus, we need new detectors that can quickly and accurately distinguish benign from malicious JavaScript inputs at scale, even when heavily obfuscated (i.e., obfuscation should not be confused with maliciousness).

While obfuscation foils techniques directly relying on content-matching, static features, such as structural constructs, can still be identified. We believe, in particular, that benign and malicious JavaScript samples have a different syntactic structure, even when both have been obfuscated. This way, machine learning-based systems would be able to leverage such purely static features, representative of the original file syntax, to distinguish benign from malicious JavaScript instances automatically. The previous assumption thereby motivates our first research question, namely *RQ1: To what extent can we detect malicious (obfuscated) JavaScript inputs by combining an analysis at the AST (Abstract Syntax Tree) level with machine learning algorithms?*

By construction, the AST solely represents the syntax of a program, though. Specifically, this code abstraction only relies on the syntactic order of the code, i.e., arbitrary sequencing choices made by the JavaScript programmers. Also, it does not have any information regarding variable dependencies or execution path conditions, e.g., dead code may be linked to a program's functionality. These shortcomings lead to the following research questions *RQ2: Can we add more semantic information into the AST of JavaScript files? Specifically, to what extent can we statically enhance the AST with control and data flows? Which features, combined with machine learning algorithms, work best to detect malicious JavaScript instances?*

While learning-based detectors may be popular to recognize new malicious JavaScript variants accurately, they are susceptible to the traditional flaws induced by machine learning approaches. Specifically, it has been shown that attackers with specific and internal knowledge of a target system may be able to produce input samples that are misclassified [65, 72, 127, 129, 183, 190]. In practice, the assumption of strong attackers does not appear realistic, as it implies access to insider information or, at least, access to a target system. At the same time, we lack investigations regarding the possibility of bypassing several learning-based detectors without needing any prior knowledge about them. This limitation then paves the way to the following research questions *RQ3: Can we present a generic attack against static malicious JavaScript detectors? More specifically, to what extent and how could attackers rewrite the ASTs of malicious JavaScript samples to reproduce existing benign ASTs while keeping the original malicious semantics? How effective would this camouflage be against static detectors?*

In practice, though, malicious JavaScript is not the only way to perform malicious activities on a victim's device. Attackers can indeed leverage more stealthy, yet powerful, attack vectors, such as vulnerable browser extensions, leading to, e.g., arbitrary code execution in a victim's browser or sensitive user data exfiltration. In fact, browser extensions have, by design, access to security- and privacy-critical APIs to perform tasks that web applications cannot traditionally do. For example, to be effective, an ad-blocker needs to read/write data on any web page or intercept network requests. Also, and contrary to JavaScript served on web pages, extensions can download arbitrary files and access arbitrary cross-domain data, even when a user is logged in.

Due to their high privileges, extensions naturally attract the interest of attackers [1, 88, 102, 213, 218]. Still, for the most popular desktop browser Chrome, with a market share of 70% [195] and a gallery of 200,000 extensions, totaling 1.2 billion installs [60], Google engineers are actively working on detecting such malicious extensions in their store. In February 2020, they removed 500 extensions that were exfiltrating user data [104]. Similarly, in April 2020, they removed 49 additional extensions that were hijacking users' cryptocurrency wallets [115] and, in June 2020, an extra 70 spying extensions [132]. In addition, before being published, extensions are reviewed by Chrome vetting system to flag extensions requiring many or powerful privileges for further analyses [43] and directly detect the ones that may contain or spread malicious software. This process makes it harder to have malicious extensions in the store today.

Still, malicious extensions represent only a fraction of the extensions that may lead to security or privacy issues. In fact, attackers can also abuse vulnerable extensions to elevate their privileges through the capabilities of an extension. To this end, attackers can leverage an extension's communication channels to send payloads to this extension, tailored to exploit its vulnerabilities. Such vulnerabilities can lead to, e.g., arbitrary code execution in an extension context. Due to their inherently benign intent, these vulnerable extensions are more challenging to detect than malicious ones, e.g., as they are not doing anything suspicious. Furthermore, while they do require powerful privileges, their benign nature allows them to pass the review process. While some previous works have focussed on vulnerable extensions, they were either purely formal [26], specific to the deprecated Firefox XPCOM [154] infrastructure [12, 25], or based on primarily manual analysis [29]. To the best of our knowledge, only EmPoWeb from Somé [186] focuses on analyzing extensions' susceptibility to attacks through external messages at scale. Still, his work is based on a lightweight call graph analysis, yielding an extremely high number of reports to vet: out of the 3,300 extensions he flagged, only 5% were vulnerable.

Given the large volume of extensions in the wild, we cannot manually review all of them to check for vulnerabilities. Therefore, we need an approach to automatically and accurately detect such vulnerable extensions at scale. In fact, we currently lack a precise analyzer to limit the number of extensions falsely reported as vulnerable to reduce the manual validation effort. To this end, we are investigating the following research question *RQ4: To what extent and how can we statically analyze browser extensions to detect suspicious data flows from and toward security- and privacy-critical APIs?*

1.2 Contributions

This work consists of four major parts, in each one of which we answer a research question. We first present JaSt, which combines a traversal of the AST with machine learning algorithms to automatically and accurately detect malicious JavaScript samples (*RQ1*). Second, we propose JStap, which extends the detection capabilities of existing lexical and AST-based pipelines by leveraging control and data flow information. This way, JStap goes beyond relying on the sole code syntax by also considering semantic information for a more accurate malicious JavaScript detection (*RQ2*). As JaSt and JStap rely on machine learning algorithms to distinguish benign from malicious JavaScript instances, they could be impacted by adversarial attacks specifically tailored for them. Third and with HideNoSeek, we go one step further and present a generic attack against AST-based malicious JavaScript detectors. To this end, we automatically rewrite malicious JavaScript inputs so that they have the same AST as existing benign scripts while retaining their original malicious semantics (*RQ3*). While we previously focussed on JavaScript instances, which are inherently benign or malicious, attackers can also leverage benign-but-buggy code to perform malicious activities. To this end, and given their high privileges, browser extensions are a target of choice for malicious actors. Finally, with DoubleX, we propose a purely static approach to detect suspicious data flows between attackers and security- and privacy-critical APIs in browser extensions (*RQ4*).

1.2.1 Detecting Malicious JavaScript Through AST Analysis (*RQ1*)

JaSt is a fully static pipeline to detect malicious JavaScript instances automatically. As malicious samples are obfuscated to hide their malicious intent, while benign scripts may be obfuscated to protect their intellectual property, we abstract JavaScript source code at a syntactic level over the AST for our analysis. This way, we eliminate the artificial noise induced, e.g., by identifier renaming, to focus solely on the structural constructs, which differ between benign and malicious (obfuscated) samples.

Specifically, JaSt traverses the AST to extract syntactic units. To preserve the units' context, e.g., a `for` loop or a `try/catch` block, we extract 4-grams whose frequency is analyzed and processed by a random forest classifier. We find that these features differ between benign and malicious JavaScript samples, allowing our classifier to learn to automatically distinguish them.

In practice, we evaluate JaSt on an extensive dataset of over 105,000 samples, where it yields a high detection accuracy of almost 99.5% and has a low false-negative rate of 0.54%, thereby outperforming related work. Following that, we discuss the evolution of our accuracy over time. Finally, we leverage different classes of malicious JavaScript samples, e.g., emails vs. exploit kits, to showcase that syntax-based features are core in classifying JavaScript inputs.

1.2.2 Improving the Detection with Semantics in the AST (*RQ2*)

Nevertheless, JaSt is not infallible and still leads to misclassifications. In fact, it lacks semantic information to go beyond solely relying on the code syntactic structure. For this purpose, we propose JStap, a modular static malicious JavaScript detection

system. Contrary to JaSt, JStap goes beyond the code syntax by considering control and data flow information. This way, we do not only take the *syntactic order* of the code into account anymore but also leverage the *semantic order* of the code logic. In particular, our detector is composed of ten modules, including five ways of abstracting JavaScript code, with differing level of context and semantic information (i.e., from a lexical analysis to a control and data flow-based approach), and two ways of extracting features (i.e., n-grams vs. considering node value information). Based on the frequency of these features, we train a random forest classifier for each module.

In practice, JStap has an accuracy of almost 99.5% and outperforms JaSt and the other existing systems, which we reimplemented and tested on our dataset totaling over 270,000 samples. To further improve our detection accuracy, we combine the predictions of several JStap's modules. Such a pipeline enables us to classify almost 93% of our dataset with a detection accuracy of 99.73% and a remaining 6.5% with an accuracy still over 99%, meaning that less than 1% of our initial dataset would require additional scrutiny.

All in all, the high accuracy of our approaches showcases that malicious and benign samples have a different code syntactic structure (including differing control and data flows).

1.2.3 Camouflaging Malicious JavaScript in Benign ASTs (*RQ3*)

While JaSt and JStap leverage differences in the code syntax for an accurate malicious JavaScript detection, it also implies that malicious samples mimicking a benign syntactic structure would, by construction, be misclassified by our machine learning-based pipelines. To this end, previous attacks against learning-based detectors leveraged specific and internal knowledge of a target system to manipulate malicious samples, e.g., such that they mimicked a benign feature distribution, for them to specifically evade a targeted system. With HideNoSeek, we go one step further and propose a generic camouflage attack, which evades the entire class of detectors based on syntactic features, without needing any information about the systems it is trying to bypass.

Our attack consists in automatically rewriting the ASTs of malicious JavaScript files into existing benign ASTs while keeping the original malicious semantics. In particular, HideNoSeek uses malicious JavaScript payloads and searches for similarities at the AST level between the payloads and traditional benign scripts. Specifically, HideNoSeek replaces benign sub-ASTs with identical malicious ones and adjusts the benign data flows–without changing the AST–so that the malicious semantics is kept after execution.

In practice, we leverage 23 malicious JavaScript payloads to generate 91,020 malicious samples, which perfectly reproduce ASTs of Alexa top 10k websites. Besides syntactic detectors, we finally showcase that our attack also evades lexical, control flow, and data flow-based classifiers, which have false-negative rates between 99.95% and 100% on our crafted inputs.

1.2.4 Statically Analyzing Browser Extensions at Scale (*RQ4*)

While JaSt and JStap aim at detecting JavaScript instances, which are inherently malicious, other samples may be benign by nature but still able to perform malicious

activities in a specific context. Due to their high privileges, browser extensions are then a target of choice for malicious actors. In fact, web pages and extensions may be isolated, but they can still communicate through messages. Specifically, a vulnerable extension can receive messages from another extension or web page.

To automatically detect if attackers could leverage these communication channels to gain access to an extension's privileges, we propose our static analyzer DoubleX. DoubleX abstracts extension code with a graph, including control and data flows, pointer analysis, and models the message interactions within and outside of an extension. This way, we track and detect suspicious flows between external actors and dangerous APIs, which can lead to, e.g., arbitrary code execution or sensitive user data exfiltration.

In practice, we evaluate DoubleX on over 154,000 Chrome extensions where it flags 309 data flows (in 278 extensions) as suspicious. We further verify that 89% of the reported flows can effectively be influenced by external actors and demonstrate exploitability for 209 of them (184 extensions). To limit the number of vulnerable extensions in the wild, we finally envision that DoubleX could be added to Chrome extension vetting system.

1.3 Publications and Tools

The contributions presented in this thesis are based on different papers. In this section, we reference these papers as well as their corresponding open-source implementations. We then discuss design choices of the original papers and the extent to which they influence this thesis.

Publications and Submissions

This thesis is based on the following papers (Chapters 3-6), which have–as of June 2021–all but one been accepted at peer-reviewed conferences:

[P1] Fass, A., Krawczyk, R. P., Backes, M., and Stock, B. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In: *DIMVA*. 2018.

[P2] Fass, A., Backes, M., and Stock, B. JStap: A Static Pre-Filter for Malicious JavaScript Detection. In: *ACSAC*. 2019.

[P3] Fass, A., Backes, M., and Stock, B. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In: *CCS*. 2019.

[P4] Fass, A., Somé, D. F., Backes, M., and Stock, B. DoubleX: Statically Analyzing Browser Extensions at Scale. In: *Under submission*. 2021.

In addition to the papers presented in this thesis, we have co-authored one more paper, namely *Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild*, to be published at DSN 2021 [S1]. In this paper, we study at large scale which code transformation techniques are used in the wild, both by benign and malicious JavaScript samples, as well as their evolution over time. In particular, we highlight the differences between popular benign transformation techniques, which are identical between client-side and library-based JavaScript, and the prevalent malicious ones.

Open-Source Implementations

For this thesis, we developed different research prototypes to evaluate and test our approaches. The source code of all our systems (and reimplementations) is available online. In particular, we developed the following tools:

- JaSt: our AST-based approach to detect malicious JavaScript inputs [T1];
- JStap: our modular system (including lexical, syntactic, control flow, and data flow analyses) to detect malicious JavaScript inputs [T2];
- HideNoSeek: our control and data flow-based approach to detect syntactic clones between two JavaScript inputs (for ethical reasons, we chose not to publish the core of our attack, i.e., the replacement of a benign AST by an equivalent malicious one) [T3];
- DoubleX: our static analyzer, which provides a semantic abstraction of browser extension source code (based on control, data, and message flows and pointer analysis) to detect suspicious data flows from and toward specific APIs (we will make our tool available on GitHub after the paper acceptance, in the meantime, we host our research prototype on Dropbox) [T4];
- Our reimplementation of Cujo [T5];
- Our reimplementation of Zozzle [T6].

Remarks

The following Chapters 3 to 6 are based on the four previously highlighted papers and represent research work conducted between 2017 and 2020. In particular, for this thesis, we keep the methodology and evaluation as presented in the corresponding (peer-reviewed) papers. For example, our datasets may vary between the different chapters. In fact, at the time of the papers' writing, we had samples, e.g., from different sources or collected at different time-frames. We also made different design choices regarding, e.g., a maximum sample size or a specific timeout.

In this thesis, we compare our approaches, among others, with the academic work Cujo [173] and Zozzle [50]. For JaSt (Chapter 3), we compared our results directly with the corresponding papers. For HideNoSeek (Chapter 5), we had to reimplement Cujo and Zozzle to test our crafted samples on them; this way, we could showcase that our attack is effective in practice. As we implemented JStap (Chapter 4) after HideNoSeek, we had the opportunity to use our reimplementations of Cujo and Zozzle to directly compare our detection performance with theirs.

Finally, we could not evaluate HideNoSeek with JStap in the corresponding papers due to concurrent submissions. For this thesis, though, we performed this evaluation, and we discuss the results in Section 5.4.

1.4 Outline

This thesis is structured into eight chapters. In this Chapter 1, we presented our motivation for conducting the work described in this thesis before introducing our contributions. Next, in Chapter 2, we give an overview of the technical background necessary to fully understand this thesis. Following that, in the next four chapters, we present our contributions to answer our four research questions. Specifically, Chapter 3 introduces our AST-based approach to accurately distinguish benign from malicious JavaScript inputs. Subsequently, in Chapter 4, we go beyond the AST by also considering control and data flow information to further detect malicious JavaScript files. On the contrary, Chapter 5 proposes a generic attack to evade both our malicious JavaScript detectors as well as state-of-the-art static classifiers. In the following Chapter 6, we move from a benign vs. malicious JavaScript analysis to an orthogonal approach revolving around vulnerable JavaScript code from browser extensions. In Chapter 7, we then present related work and highlight the added value of our contributions. Finally, in Chapter 8, we discuss future work before summarizing our contributions and concluding this thesis.

2
Technical Background

In this chapter, we present the technical background relevant to this thesis. First, we introduce the scripting language JavaScript and highlight differences between benign and malicious JavaScript samples. We then present different techniques to perform a static analysis of JavaScript files. To automate this process and discover meaningful patterns in benign vs. malicious JavaScript inputs, we subsequently discuss machine learning techniques, specifically the classification and clustering strategies. Finally, we present the browser extension ecosystem, with a focus on its security architecture and the communication within and outside of an extension.

2.1 JavaScript

JavaScript is a browser scripting language that was designed to enhance the interactivity of websites and to improve their user-friendliness. We first underline the origin of the language before giving an overview of additional Web technologies relevant to understand this thesis. Due to the popularity of JavaScript, it also attracts the interest of malicious actors. We then highlight the difference between vulnerable (benign) and malicious JavaScript, present different malicious JavaScript categories, and explain how malicious JavaScript is spread to harm Internet users. Finally, we discuss different techniques to transform JavaScript code to make it harder to understand and analyze.

2.1.1 Origin of JavaScript

JavaScript was designed at Netscape in 1995 to make web pages more dynamic. It was subsequently released in the beta version of the Netscape Navigator 2.0 browser, under the name *LiveScript*, before being changed to *JavaScript* [5]. With the popularity of the language, Microsoft released *JScript* in 1996, its implementation of JavaScript, in the Internet Explorer 3.0 browser. Following that, ECMA International developed the first JavaScript standard, which was adopted in 1997. While the name of the language specification is *ECMAScript*, the commonly used term stays *JavaScript* [56].

Nowadays, JavaScript has become one of the core technologies of the Web platform. As of October 2020, it is used by almost 97% of the websites as a client-side programming language [212]. Also, with *Node.js* [93], JavaScript can be used for server-side scripting as well as to write command-line tools and applications.

2.1.2 JavaScript Interfaced to the Browser

In this section, we focus on JavaScript in the context of the web browser. Specifically, we define the Web concepts necessary to understand the remaining of this thesis.

The most basic building block of the Web is HTML (HyperText Markup Language) [221], which is the standard markup language to parse and display documents in a web browser. In practice, a web server sends a static HTML document to the web browser, which renders it into web pages. To improve the pages appearance, HTML can be combined with CSS (Cascading Style Sheets). Besides, to enable dynamic interactions with web pages, make them more sophisticated and user-friendly, HTML documents can embed programs written in JavaScript. In practice, JavaScript cannot interact with web pages on its own but needs to leverage the DOM (Document Object Model) API [220].

In fact, when an HTML document is parsed, the browser stores its structure as a logical tree. The DOM then enables JavaScript to access and modify the document structure, style, and content.

The combination of HTML, DOM, and JavaScript enables programmers to create dynamic web applications. For security and privacy reasons, web pages should not trust each other, hence the need for isolation mechanisms. To this end, the Same Origin Policy (SOP) [147] ensures that only resources with the same origin (i.e., same scheme, host, and port) [222] are allowed to interact with each other. Nevertheless, there are some ways to relax the SOP, such as postMessages [78], which enable web content from different origins to communicate with each other. In this case, the responsibility to prevent cross-origin attacks is partially delegated from the web browser to the developers, who should only handle postMessages coming from origins they trust.

2.1.3 Malicious JavaScript

Still, the previous mechanisms are not sufficient to guarantee the security and privacy of the Web users. In fact, the ubiquity of JavaScript, combined with its execution directly in the users' browser, renders it attractive for attackers too.

In this thesis, we make a distinction between *benign-but-buggy* JavaScript, i.e., vulnerable code designed by well-intentioned developers, and *malicious* JavaScript samples, which have been designed by malicious actors with the aim of harming victims. For example, vulnerable code includes a web page (or browser extension) executing untrusted data without proper sanitization, which leads to Cross-Site Scripting (XSS) attacks [106]. To mitigate such threats, the Content Security Policy (CSP) [219] can be deployed to limit (or ban) the execution of untrusted scripts. While we focus on benign-but-buggy JavaScript code in Chapter 6, Chapters 3-5 consider malicious JavaScript instances, which we describe in the following.

In this thesis, we use the term *malicious JavaScript* to both refer to JavaScript code directly performing malicious activities (e.g., ransomware written in JavaScript) as well as JavaScript payloads leading to the launch of the actual attacks against victims' machines. For the latter, such payloads generally trigger the download of additional malware [169] (e.g., ransomware or banking trojans); thus, such attacks are usually referred to as *drive-by downloads*. The most popular way to launch these attacks relies on exploit kits, which stealthily probe a victim's machine for vulnerabilities they could exploit, e.g., targeting specific browsers versions or plugins. After successfully exploiting a vulnerability, the next infection stage begins with, e.g., the download and execution of the actual malware in the victim's environment. Besides exploit kits, malicious JavaScript payloads can be droppers. Similarly, upon execution (such as after a double click on a dropper file), they will also lead to the next infection stage.

There are numerous ways to spread malicious JavaScript in the wild and infect users' machines. For example, there are two popular ways to perform a drive-by download attack. The victim either visits a malicious or compromised web page (which can then probe the system for vulnerabilities) or opens a malicious email attachment. To lure a user into visiting a malicious web page or opening a malicious attachment, attackers use social engineering techniques. In addition, a user can also get redirected to a

malicious website, e.g., after having clicked on a malicious advertisement found on a legitimate website.

2.1.4 JavaScript Code Transformation Techniques

To impede the detection and analysis of their malicious JavaScript samples, malware authors abuse obfuscation techniques. Nevertheless, obfuscation should not be confused with maliciousness. By design, obfuscation transforms the code to make it harder to understand, both for human analysts and automatic tools. In practice, benign code can also be obfuscated, e.g., to protect code privacy and intellectual property. Several categories of code obfuscation techniques can be found in the wild [84, 98, 101, 226]:

- **Randomization obfuscation** consists in randomly inserting or changing elements of a script without altering its semantics. It includes adding whitespace characters (*whitespace randomization*) or randomly inserting comments (*comment randomization*), which makes the code harder to read and could foil techniques relying on content matching. In addition, variable and function names can be replaced with randomly created strings with no specific meaning (*identifier obfuscation*), e.g., to hinder manual analysis.
- **Data obfuscation** regroups string and number manipulation techniques. For example, strings can be split, concatenated, or reversed to not appear in plain text. Similarly, characters can be substituted, e.g., by running a regular expression replace on a string. Also, standard or custom encoding (such as ASCII, Unicode, or hexadecimal), encryption and decryption functions hinder a direct understanding of the code. These techniques are more specifically referred to as *string obfuscation*. Similarly, to avoid that a number appears in plain text, it can be, e.g., computed with arithmetic operators (*integer obfuscation*). Further techniques also enable to hide data. For example, to access the property `prop` of an object `obj`, the bracket notation `obj["prop"]` can be privileged over the dot notation `obj.prop` (*obfuscated field reference*). In fact, the bracket notation enables to compute an expression (e.g., window["e" + "val"]), whereas the dot notation only considers identifiers [146]. In addition, data can be fetched from a global (dynamic) array to complicate the understanding of the code. For this purpose, data can also be rewritten, so that it does not contain, e.g., any alphanumeric character anymore [157].
- **Logic structure obfuscation** directly targets the code logic, such as manipulating execution paths, e.g., by adding conditional branches. Additional techniques consist in adding irrelevant instructions (*dead code injection*) or changing the program flow, e.g., by moving all basic blocks in a single infinite loop, whose flow is controlled by a `switch` statement [94] (*control flow flattening*).
- **Dynamic code generation** leverages the dynamic nature of JavaScript to generate code on the fly, e.g., with `eval`. Specifically, this construct can be used as a *packer* function, e.g., to evaluate encoded code [57, 126].
- **Environment interactions** is specific to web-JavaScript. In this case, statements can be split and scattered across an HTML document using multiple `< script >` blocks. Similarly, the payload can be stored within the DOM and extracted subsequently so that it is not directly discernible.

Obfuscation should also not be confused with minification, which aims at reducing the code size, without aggressively attempting to hide its functionality. Basic techniques consist of deleting whitespaces and comments (while obfuscation may randomly add them to impede the analysis), shortening variable names, and removing dead code. More advanced techniques directly modify the logic structure of the code, e.g., by eliminating unreachable or redundant code, inlining functions [70, 135], or replacing `if` statements with the conditional operator shortcut [139].

2.2 Static Analysis

To detect and characterize malicious JavaScript files, we can conduct two types of analysis. Dynamic analysis consists in executing the input samples in a controlled environment, e.g., sandbox, to simulate and record their behavior. By design, dynamic analysis has the following drawbacks: it is very costly, both in terms of specific instrumentation and run-time performance; also, the amount of time required to observe a potential malicious behavior is not clearly defined [190]. In addition, dynamic analysis explores one execution path, meaning that it would not necessarily detect malicious inputs that are time- or environment-dependent [11, 23]. On the contrary, static analysis consists in analyzing input samples without executing them. As motivated in Chapter 1, this thesis relies on a static analysis of JavaScript files, mainly due to its speed, accuracy, and high code coverage. In the following chapters, we refer to different static analysis techniques, which we present in this section:

- **Content-based analysis** relies on specific patterns from the code. For example, it is used by traditional anti-virus signatures to detect malicious inputs. By design, this approach is only effective against known malware and is easily thwarted by code obfuscation techniques.
- **Lexical analysis** consists in abstracting the code at the token level. To this end, each word is linearly converted into a lexical unit (i.e., token). For example, lexical analysis does not consider variable names anymore but represents them with the generic `Identifier` token. The same applies to variable values, which can be abstracted to their types. Thus, lexical analysis is resilient to, e.g., identifier obfuscation.
- **Syntactic analysis** leverages tokens to build a tree-like representation (i.e., the Abstract Syntax Tree, AST) of the source code, which depicts the code grammatical structure [2]. Thus, the AST goes beyond tokens, as it does not represent a linear word to unit mapping anymore but abstracts the code's original nesting of programming constructs. By design, though, the AST represents programmers' arbitrary sequencing choices and does not enable to reason about the order in which statements are executed nor about conditions leading to specific execution paths.
- **Control flow analysis** relates to the construction of the CFG (Control Flow Graph). The CFG represents the program as a graph in which the nodes are statements and predicate expressions. The nodes are then connected with labeled and directed edges to represent flows of control in the program. Specifically, each statement node has an outgoing edge ϵ to indicate the order in which statements are executed. Similarly, each predicate node has two outgoing edges labeled with *true* or *false* to reason about the conditions that have to be met for a specific execution path to be taken. In this

thesis, we adopt a definition of the CFG that slightly differs from Allen's [3] as we enhance our AST with control flow edges. This way, we build a joint structure that combines control flow information with the fine-grained AST nodes and edges. We describe our resulting implementation in Section 4.1.1.3.

- **Data flow analysis** consists in reasoning about variable dependencies. This information can then be stored in the PDG (Program Dependence Graph). This graph is constructed with two types of directed edges on the statement and predicate nodes, namely control dependency edges (obtained through the combination of control flow and dominator information [2, 3], meaning that statement order information is lost) and data dependency edges. The latter represents the influence of a variable on another. Specifically, there is a data flow between two statement nodes if and only if a variable is defined at the source node and used at the destination node, with respect to the variable reaching definition. In this thesis, we adopt a definition of the PDG that slightly differs from the one of Ferrante et al. [64] as we chose to add data flow edges to our CFG. This way, we retain information regarding statement order and have a fine-grained representation of the data flows directly at the variable level (as we build the CFG upon the AST). We describe our corresponding implementations in Section 4.1.1.4 (for JStap and HideNoSeek) and Section 6.2.2.3 (for DoubleX).
- **Pointer analysis** determines which pointers can point to a given variable [2]. By design, this approach enables us to infer information regarding variable values and keep track of these values.

2.3 Machine Learning

The previously presented static analysis techniques can then be combined with machine learning algorithms to better understand and analyze complex data. In this thesis, we focus on two categories of learning, namely classification, to distinguish benign from malicious JavaScript inputs, and, to some extent, clustering, to group malicious JavaScript files that have a similar structure.

2.3.1 Classification

Classification is a form of supervised learning, meaning that it requires training data and prior information regarding this data. Specifically, classification consists in discovering meaningful patterns from labeled data, generalizing them in a model that encapsulates the relationships between instances with similar patterns, and making predictions on future data [174, 224]. By design, a classifier pipeline would enable us to, e.g., learn specific patterns typical of known benign vs. malicious JavaScript samples (for example, based on lexical or syntactic features) and recognize similar patterns in previously unseen inputs.

Formally, we define $\phi : \mathcal{X} \to \mathbb{R}^n$, which maps a JavaScript sample $x \in \mathcal{X}$ to its n-dimensional feature vector. We consider σ, our model that encapsulates the properties of benign vs. malicious JavaScript samples. To make predictions on unknown data, we define $\psi_\sigma : \mathbb{R}^n \to \mathcal{P}$, which maps a feature vector $v \in \mathbb{R}^n$ to an output space $\mathcal{P}$, and $f_\sigma = \psi_\sigma \circ \phi$, which maps a JavaScript sample $x \in \mathcal{X}$ to $\mathcal{P}$. For a sample $x \in \mathcal{X}$,

$f_\sigma(x)$ represents the outcome of the classifier for this sample. This outcome can be directly the classifier prediction, i.e., $\mathcal{P} = \{0, 1\}$, where 0 represents the benign class and 1 the malicious one. Alternatively, $f_\sigma(x)$ can also quantify the probability $y_0(x)$ of x being benign and $y_1(x)$ of x being malicious, i.e., $f_\sigma(x) = (y_0(x), y_1(x))$, with $y_0(x) + y_1(x) = 1$ and $\mathcal{P} = \mathbb{R}^2$. Traditionally, a classifier returns the class label with the highest probability. We can also define a custom threshold $t \in [0, 1]$ so that, e.g., for a sample $x \in \mathcal{X}$, if $y_1(x) \geqslant t$, the output label should be 1. To achieve an optimal trade-off between the samples wrongly classified as malicious (false positives) and those wrongly classified as benign (false negatives), this threshold t can be determined by using Youden's J statistic [168, 232]. We give an example in Section 3.2.2.2.

In practice, to learn from labeled benign and malicious inputs and make predictions on unknown samples, different classification algorithms can be used. In this thesis, we refer to the following ones:

- **Bernoulli naive Bayes** assumes naive independence between every pair of features, meaning that it can learn the parameters for each feature separately (in practice, this assumption is likely incorrect). To predict which class a given sample belongs to, this algorithm leverages the presence or absence of a feature as a boolean attribute [91].
- **Multinomial naive Bayes** makes the same independence assumption as Bernoulli naive Bayes but considers feature probability (instead of feature presence vs. absence) to make predictions [130].
- **Support Vector Machine (SVM)** draws a hyperplane in the feature space to separate the benign from the malicious class with maximum margin [167].
- **Random forest** does not assume independence between the different features considered. Random forest is a meta estimator that combines the predictions of several decision trees [22] and classifies an unknown input sample according to the decision of the majority of the tree predictors. By design, this algorithm combines two types of randomness. First, each decision tree is built from random sub-samples of the initial dataset. Second, for each tree predictor, an input is entered at the top of the tree, and as it traverses down the tree, a feature subset is randomly selected at each node to generate the best split [21].

2.3.2 Clustering

Contrary to classification, clustering is a form of unsupervised learning, meaning that we do not have any prior knowledge regarding the data (i.e., we have access to samples from $\mathcal{X}$ without having any information regarding the output space $\mathcal{P}$). By design, clustering consists in grouping elements that belong together, based on similarity algorithms or a predefined number of clusters. In Section 5.3.1.1, to cluster similar malicious JavaScript samples together, we use the *k-means++* algorithm [7]. This approach aims at minimizing the average squared distances between points (i.e., feature vectors) from the same cluster. First, k distant points in the feature space are selected as cluster centers. Next, each remaining point is assigned to the nearest cluster center, and each center is recomputed as the center of mass of all points from this cluster. Finally, the process is repeated until a fix point is reached.

This part concludes the background relevant to Chapters 3, 4, and 5.

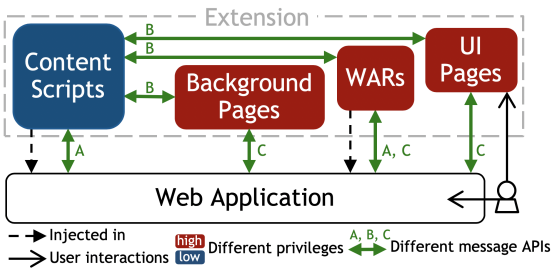


Figure 2.1: Extension architecture

2.4 Browser Extensions

In this section, we consider browser extensions, which are the focus of Chapter 6. Browser extensions are third-party programs, which users can install to extend their browser functionality, e.g., by adding ad-blocking capabilities. We first present the extension ecosystem with a highlight on security mechanisms extensions implement. Subsequently, we introduce their architecture before considering the message-passing APIs they use to communicate.

2.4.1 Presentation and Security Considerations

Browser extensions, which are zipped bundles of, e.g., HTML, JavaScript, or CSS files, are widely used to enhance users' browsing experience. While some extensions merely customize users' browser interface, others serve more security- and privacy-critical tasks, e.g., to be effective, an ad-blocker needs to modify web page content. Therefore, extensions have privileged capabilities in comparison to web applications. Unlike JavaScript code in web pages, they are not restricted by the Same Origin Policy and can access arbitrary cross-domain in the logged-in context of the user's browser and inject code into any document.

Due to their high privileges, extensions may introduce security and privacy threats and put their large user base at risk. To limit these risks, extensions only have access to explicitly declared APIs. In particular, every extension contains a `manifest.json` file, which provides some information, e.g., regarding an extension most important files or capabilities [44]. For example, there is a *permissions* field that lists the different APIs and hosts a given extension has access to [39, 144]. Such permissions include the possibility for an extension to read and write user data on any or specific web pages (`host`), to store and retrieve data from the extension storage (`storage`), to download arbitrary files (`downloads`), and to access users' history (`history`).

2.4.2 Architecture

As represented in Figure 2.1, an extension is divided into four main components:

- **Content scripts** are injected by an extension to run along with web applications. These content scripts can use the standard DOM APIs to read and modify web pages and have access to `localStorage` (i.e., storage space for a document origin, saved across browser sessions) [152], similarly to the scripts loaded by web pages.

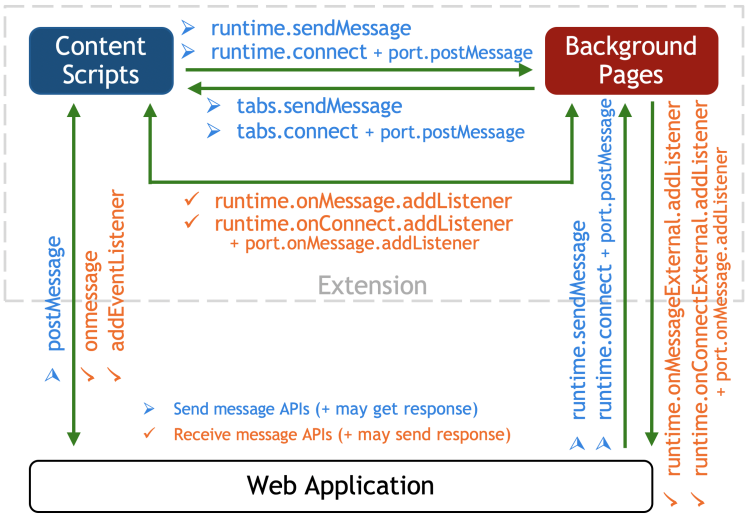


Figure 2.2: Extension message-passing APIs

- **Background pages** contain the extension core logic and run independently of the lifetime of any particular web page or browser window.
- **UI pages** enable users to customize the extension behavior, e.g., over different options, settings, or pop-ups.
- **WARs (Web Accessible Resources)** list additional resources that the extension needs to expose to web applications, e.g., scripts to be executed on every page.

Each component is composed of several files, which are specified in the extension manifest [41]. While the highly-privileged background pages, UI pages, and WARs have access to the full extension's capabilities (as specified in the manifest *permissions* field), the less privileged content scripts only have access to the `host` (at least until Chrome Manifest V3 is deployed [46]) and `storage` permissions.

2.4.3 Communication Channels

By design, an extension can communicate with web pages and other extensions. In this section, we present the communication channels between a web page and each extension component, within an extension, and between two extensions.

Web Application - Content Script Besides the web application DOM and `localStorage`, content scripts and web pages communicate over messages. As represented in Figure 2.2, both use the `postMessage` API [153] to send messages, and `addEventListener` [140] or `onmessage` [151] to receive them. We give an example in Listing 2.1, where the web application sends a message, which the content script receives (as in Figure 2.2, the APIs to send and receive messages are represented in blue and orange, respectively). By default, the content scripts receive all messages sent toward the window in which they are injected. Thus, if a page running a content script receives a `postMessage` from another page, the content script's handler is also invoked.

```
1 // Web application code
2 window.postMessage("Hi CS", "*"); // Sends
3
4 // Content script code
5 window.onmessage = function(event) {
6  received = event.data; // received = "Hi CS"
7 }
```

Listing 2.1: Messages: web application - content script

```
1 // Content script code
2 chrome.runtime.sendMessage({greeting: "Hi BP"}, function(response) {
3  received = response.farewell; // received = "Bye CS"
4 });
5
6 // Background page code
7 chrome.runtime.onMessage.addListener(
8  function(request, sender, sendResponse) {
9   received = request.greeting; // received = "Hi BP"
10  sendResponse({farewell: "Bye CS"});
11 });
```

Listing 2.2: Messages: content script - background page (one-time)

Content Script - Background Page There are two types of APIs to exchange messages between a content script and a background page. The *one-time requests* API aims at sending a single message and receiving a response, while *long-lived connections* leverage an established message port and stay open to exchange multiple messages [45]. As presented in Figure 2.2, the content script uses `runtime.sendMessage` (one-time) or `runtime.connect` (long-lived) to send messages.[1] Similarly, the background page sends requests with `tabs.sendMessage` (one-time) or `tabs.connect` (long-lived). Both for the content script and background page, for Chromium-based extensions, the last parameter of these messaging APIs can be a callback to get the response sent by the other component. For Firefox, these APIs can return a `Promise` [145], instead of invoking a callback, so that the `.then` construct can be used to handle the response. As for receiving messages (and responding), both components register a listener: `runtime.onMessage.addListener` (one-time) or `runtime.onConnect.addListener` (long-lived). Listing 2.2 illustrates Chrome one-time requests API: the content script sends the message *{greeting: "Hi BP"}*, which the background page receives before responding *{farewell: "Bye CS"}* back.

Web Application - Background Page For Chromium-based extensions, a web application and a background page can directly communicate under two assumptions [45]. First, the extension should fill the `externally_connectable` field from its manifest

[1] For legibility reasons, we omit `browser/chrome` from the APIs, which would be, e.g., `chrome.runtime.sendMessage` or `browser.runtime.sendMessage`

```
1 // Web application code
2 var port = chrome.runtime.connect({name: "myport"});
3 port.postMessage({greeting: "Hi BP"});
4
5 // Background page code
6 chrome.runtime.onConnectExternal.addListener(function(p) {
7  p.onMessage.addListener(function(message) {
8   received = message.greeting // Hi BP
9  });
10 });
```

Listing 2.3: Messages: web application - background page (long-lived)

with specific URLs, to allow the communication with the corresponding web pages only. Second, the communication can only be initiated by the web application. As presented in Figure 2.2, the web application sends requests (and gets a response) with `runtime.sendMessage` (one-time) or `runtime.connect` (long-lived). The background page receives messages (and responds) with `runtime.onMessageExternal.addListener` (one-time) or `runtime.onConnectExternal.addListener` (long-lived). Listing 2.3 illustrates Chrome long-lived connections.

Case of UI Pages and WARs Like the background pages, UI pages and WARs are part of the extension core. To exchange messages with the content scripts, they use the same APIs as those used by the background pages. To communicate with the background pages, they can directly leverage their shared extension storage. To exchange messages with a web application, UI pages and WARs use the same APIs as those used by the background pages. As the WARs can be injected as iframes in web pages, they also leverage the same APIs as those used by the content scripts to interact with a web application. Figure 2.1 summarizes the three sorts of messaging APIs that extension components use, while Figure 2.2 presents the specific APIs.

Extension A - Extension B Finally, two extensions can communicate with one another. In this case, the message-passing APIs are the same as those for the communication between a background page and a web application. Still, this time, the communication is enabled *by default* with all extensions [42]. To interact with specific extensions only, an extension must explicitly declare the IDs of allowed extensions in its manifest.

2.5 Summary

In this chapter, we laid the technical background for the remainder of this thesis. We first introduced the scripting language JavaScript and highlighted its usage in the context of the web browser. We pointed out differences between benign and malicious JavaScript and underlined some techniques to transform JavaScript code to make it harder to analyze. Subsequently, we presented different approaches to abstract and analyze JavaScript code statically. To automate the analysis and distinguish benign

from malicious JavaScript inputs, we then introduced the classification and clustering concepts from the machine learning ecosystem. Finally, we considered browser extensions with a focus on the security risks they may induce. We presented their architecture and components before discussing the message-passing APIs they can use to communicate.

In the following chapter, we combine a static analysis of JavaScript files with machine learning to automatically detect malicious JavaScript instances.

3

JAST: An AST-Based Malicious JavaScript Detector

Given the popularity of the Web platform, it naturally also attracts the interest of malicious actors. Specifically, attackers abuse JavaScript to, e.g., exploit bugs in the browser or perform drive-by downloads. Due to the large volume of JavaScript files in the wild, executing all of them to check for maliciousness is not realistic. At the same time, to hinder the analysis and the detection of such nefarious scripts, malicious actors take advantage of code obfuscation. Even though it foils techniques directly relying on content matching, e.g., signatures, it leaves specific and recurrent traces in the code syntax, which static systems can process.

In this chapter, we answer *RQ1: To what extent can we detect malicious (obfuscated) JavaScript inputs by combining an analysis at the AST (Abstract Syntax Tree) level with machine learning algorithms?* To this end, we present JaSt, our fully static pipeline to automatically distinguish benign (obfuscated) from malicious (obfuscated) JavaScript instances. JaSt first abstracts JavaScript code at a syntactic level over the AST. This way, we eliminate the artificial noise induced, e.g., by identifier renaming, to solely focus on the programmatic and structural constructs, which differ between benign and malicious scripts. In particular, JaSt learns to recognize specific syntactic patterns either typical of benign or of malicious JavaScript inputs. In practice, we evaluate the detection performance of our approach on 105,305 unique files and underline our higher accuracy compared to prior work. We then discuss the applicability of JaSt in the wild, e.g., in terms of samples evolution over time and syntactic similarities between different classes of malicious JavaScript.

3.1 Learning from AST-Based Features

To detect malicious JavaScript samples at scale, JaSt performs a static analysis over the AST. In particular, we traverse the AST to extract syntactic units (Figure 3.1 stage 1). To preserve the units context, e.g., a `for` loop or a `try/catch` block, we subsequently extract substrings of length n, namely n-grams, whose frequency we analyze (Figure 3.1 stage 2). We find that these features differ between benign and malicious samples, enabling our random forest classifier to learn to automatically distinguish them (Figure 3.1 stage 3).

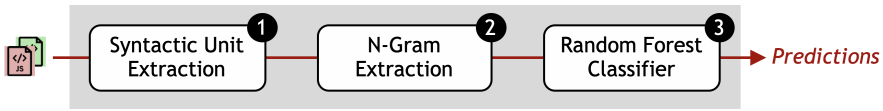


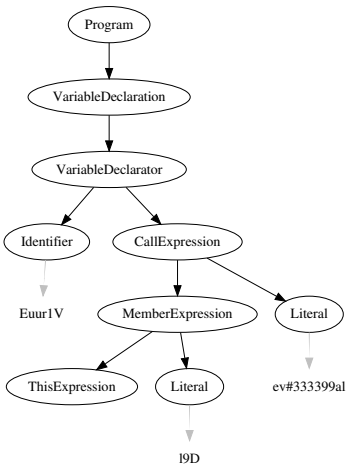
Figure 3.1: Architecture of JaSt

3.1.1 Syntactic Analysis

JaSt leverages the parser Esprima [79] to generate the AST of a valid JavaScript sample. We chose to rely on Esprima given its thorough set of test cases [80] and widespread use by prior work [25, 74, 120, 158, 163, 188, 191, 197]. Even though we designed JaSt in Python, we invoke the *Node.js* implementation of Esprima, rather than the Python port, for performance reasons. As presented in Section 2.2, the AST is an

```
1 var Euur1V = this [ "l9D" ] ("ev#333399al")
```

(a) Malicious JavaScript example from (198)



(b) AST from (a)

Syntactic unit	Simplified name
Identifier	Identifier
ThisExpression	Expression
Literal	Literal
MemberExpression	Expression
Literal	Literal
CallExpression	Expression
VariableDeclarator	Declarator
VariableDeclaration	Declaration
Program	Program

(c) Syntactic units extracted from (b)

Figure 3.2: AST generation and corresponding syntactic unit extraction

ordered tree, which describes the syntactic structure of a program. We chose to perform a syntactic analysis to detect malicious JavaScript instances due to its capability to quickly and accurately analyze obfuscated inputs. While, at textual level, obfuscation can foil the detection of malicious files, at AST level, we can still identify structural constructs, which differ–despite obfuscation–between benign and malicious inputs [S1]. In addition, the AST provides a certain level of code abstraction, ignoring, for example, the variable names to consider them as `Identifier` nodes and skipping blank spaces. Therefore, leveraging the arrangement of syntactic units (e.g., statements, expressions, or declarations) in a given JavaScript file enables us to capture the salient properties of the code and hence to identify specific and recurrent malicious (or benign) patterns.

Next, we traverse the AST depth-first post-order to extract syntactic units. Figure 3.2 illustrates the parsing process, where the malicious entity from Figure 3.2a is transformed into an AST (Figure 3.2b), whose traversal gives a sequence of syntactic units (Figure 3.2c). Overall the parser Esprima can produce 69 different syntactic entities ranging from `FunctionDeclaration` to `ImportDefaultSpecifier`. For performance reasons, JaSt performs a simplification of the list of syntactic units returned by the parser. This process consists in grouping elements with the same abstract syntactic meaning. For example, we refer both to `FunctionDeclaration` and `VariableDeclaration` as `Declaration`. Similarly, we refer to `Statement` both for `ForStatement` and `WhileStatement`. In addition, we consider a one-element family if we could not group it with other entities (e.g., `Identifier` or `Program`). This process enables us to reduce the number of different units from 69 to 19. Still, we preserve their original syntactic meaning, as we analyze each element within its context, by using n-gram features.

n value	#All possible n-grams	#Selected n-grams	Feature reduction (%)
$n = \mathbf{1}$	19	17	10.53
$n = \mathbf{2}$	361	114	68.42
$n = \mathbf{3}$	6,859	570	91.69
$n = \mathbf{4}$	130,321	2,457	98.11
$n = \mathbf{5}$	2,476,099	8,025	99.68

Table 3.1: Number of all possible n-grams vs. number of n-grams selected

3.1.2 N-Gram Frequency

To identify specific patterns in JavaScript documents, JaSt moves a fixed-length window of length n over our list of simplified syntactic units. This way, we collect every sub-sequence of length n, namely n-grams. For example, the first two 3-grams extracted from Figure 3.2c are: (`Identifier`, `ThisExpression`, `Literal`) and (`ThisExpression`, `Literal`, `MemberExpression`). As shown by prior work, n-grams are a generic and effective modeling means to, e.g., detect malicious [109, 117, 173] or obfuscated [123] code, or even to identify anomalous network packets [215].

In fact, n-grams represent how the syntactic units were originally arranged in the analyzed JavaScript files. Therefore, documents sharing several n-grams with the same frequency present similarities with one another, while files with different n-grams have a more dissimilar content. As a consequence, leveraging the frequency of these short patterns contributes to determining if a given sample is either benign or malicious. To be able to compare the frequency of all n-grams appearing in several JavaScript files, JaSt constructs a vector space such that we associate each n-gram with one dimension while we store its corresponding frequency at this position in the vector R. For this mapping process, we do not consider all possible n-grams to limit the size of the vector space (initially of 19^n depending on the chosen length n of n-grams), which has a direct impact on performance. Besides, not all n-gram combinations are plausible, e.g., as the root of the AST, the `Program` unit can only be present once. Therefore, we create a set S containing n-grams we selected on the basis of their prevalence in our dataset. For this selection process, we consider the suitability criteria defined by Wressnegger et al. [225], namely *perturbation* (i.e., the expected ratio of n-grams in a benign sample that are not part of the training data) and *density* (i.e., the ratio of the number of unique n-grams in the dataset to the number of all possible n-grams). In particular, we aim at significantly reducing the *density* of our dataset while keeping an extremely low *perturbation*. This way, we would limit the number of false positives induced by unknown n-grams in benign data. For this purpose, we are only considering the n-grams present in our dataset (knowing that JaSt automatically updates its considered n-grams list whenever it encounters an unknown n-gram). This process enables to reduce the number of features by more than 90% for 3-, 4-, and 5-grams, as shown in Table 3.1.

Formally, we define the vector R (containing n-gram frequencies) by using the set $\mathcal{S}$ of n-grams JaSt considers and the set $\mathcal{X}$ of JavaScript samples to analyze such as:

$$R^{\mathrm{T}} = \{r_1, ..., r_{|\mathcal{X}|}\}$$

$$\text{knowing that } \forall i \in [\![1, |\mathcal{X}|]\!], r_i = \phi(x_i)$$

$$\text{and } \phi : x_i \longrightarrow (\phi_n(x_i))_{n \in \mathcal{S}}$$

with $\phi_n(x_i)$ the frequency of the n-gram n in the analyzed sample x_i .

As a consequence, the ϕ function maps a JavaScript file x_i to the vector space $\mathbb{R}^{|\mathcal{S}|}$ such that we set all dimensions associated with the n-grams contained in the set $\mathcal{S}$ to their frequency. To avoid an implicit bias on the length of the inputs, we normalize the frequencies, such that: $\forall i \in [\![1, |\mathcal{X}|]\!], ||r_i|| = ||\phi(x_i)|| = 1$. We then use the frequency vector R as input to the learning components.

3.1.3 Learning and Classification

The learning-based detection completes the design of JaSt. Before being able to predict if a given JavaScript sample is benign or malicious, our classifier should be trained on a representative, up-to-date, and balanced set of both benign and malicious JavaScript files. To build an initial model, update an existing one, or classify JavaScript inputs, we use the vectorial representation $R^{\mathrm{T}} = (r_i)_{i \in [\![1, |\mathcal{X}|]\!]}$ of the files $\mathcal{X}$ to analyze. We empirically evaluated different off-the-shelf classifiers (Bernoulli naive Bayes, multinomial naive Bayes, SVM, and random forest) and determined that random forest yielded the most accurate results.

We implemented JaSt in Python, and we leverage the Scikit-learn implementation of random forest to analyze JavaScript inputs [164]. To optimize the predictions of our learning-based detector, we first determined the tuple of hyperparameters yielding an optimal model (i.e., that minimizes a predefined loss function on an independent dataset). Our independent dataset was provided by the German Federal Office for Information Security (BSI) [24] and labeled according to the protocol described in Section 3.2.1. It contains 17,500 unique benign samples and as many malicious ones. To tune the optimal set of hyperparameters, we first performed random search [16] with 5-fold cross-validation on this dataset. This way, we sampled a fixed number of parameter settings from the specified distributions to narrow down the range of possibilities for each hyperparameter. In a second step, we could therefore use grid search to exhaustively test all resulting combinations with cross-validation.

We selected the following hyperparameters because their combination leads to the best trade-off between accuracy and performance. In particular, we chose 4-gram features as the length four provides the best trade-off between false positives and false negatives. In addition, we construct our random forest with 500 trees, which all have a maximum depth of 50 nodes. When looking for the best node split, we consider $\lceil \sqrt{2,457} \rceil = 50$ features and use the Gini criterion to measure the quality of a split, based on the Gini impurity [67]. In the following, we use these parameters to train, update, and test JaSt's random forest classifier.

Source	Collection	#Samples	Label	Obfuscated
Emails	2017-2018	85,059	Malicious	yes
Microsoft	2015-2018	17,668	Benign	yes
Games	N/A	2,007	Benign	no
Web frameworks	N/A	434	Benign	N/A
Atom	2011-2018	137	Benign	no

Table 3.2: Benign and malicious JavaScript datasets

3.2 Detecting Malicious JavaScript

To highlight the accuracy of JaSt, we analyze and classify over 105,000 unique samples from different sources. After presenting our datasets, we discuss our detection performance, which we subsequently compare with related work.

3.2.1 Benign and Malicious Datasets

The experimental evaluation of our approach rests on an extensive dataset mainly provided by the BSI [24]. It comprises 105,305 unique (based on their SHA1 hash) JavaScript samples. In particular, it includes 20,246 benign and 85,059 malicious JavaScript files (see Table 3.2) between 100 bytes and 1 megabyte (to retain JavaScript code with enough features to be representative of its benign vs. malicious intent, without downgrading the performance with too big a size).

Our malicious samples mainly correspond to JavaScript extracted from emails. In fact, emails are one of the most common and effective way to spread JScript-loaders, as a double-click on an attachment is by default sufficient to execute it on Windows hosts, leading to, e.g., drive-by download or ransomware attacks. JavaScript as infection vector is particularly relevant and powerful in this setting since it is especially prone to obfuscation. This way, attackers can build a unique copy of a malicious attachment for each recipient; thus, foiling classical anti-virus signatures. These samples have been labeled as malicious based on a score obtained after having been tested by twenty different anti-virus systems, the malware scanner of the BSI, and a run-time-based analysis.

As for the benign files, we consider several datasets, as underlined in Table 3.2. In particular, almost 18,000 samples come from Microsoft products (e.g., Microsoft Exchange 2016 and Microsoft Team Foundation Server 2017). As most of them are obfuscated, we can ensure that JaSt does not confuse obfuscation with maliciousness but leverages syntactic features for an accurate distinction between benign and malicious (obfuscated) inputs. For our benign dataset to be more up-to-date and representative of the JavaScript distribution found in the wild, we also include some open-source games written in JavaScript, web frameworks, and the source code of Atom [8] (we tested these samples either using the previous protocol or directly downloaded them from the developers' web pages). These extra samples extend our dataset with some new, sometimes unusual or specific (e.g., games) coding styles to show that JaSt does not mistake unseen nor unusual syntactic structures for maliciousness. We chose not to

Source	#Misclassifications	#Correct classifications	Accuracy (%)
Emails	443	81,116	99.46
Average benign	86	16,660	99.48
- Microsoft	71	14,097	99.50
- Games	10	1,997	99.52
- Web frameworks	4	430	99.03
- Atom	1	136	98.98

Table 3.3: Accuracy of JAST

consider any web-JavaScript extracted from HTML documents at this stage, though. We discuss our choice and perform a separate analysis in Section 3.3.2, where we showcase the applicability of our approach also on previously unseen web samples.

3.2.2 JAST Detection Performance

In our first experiment, we study the detection performance of JAST in terms of true-positive and true-negative rates (i.e., correct classification of the samples as malicious or as benign, respectively), false-positive and false-negative rates (i.e., misclassification of the samples as malicious instead of benign or as benign instead of malicious, respectively), and overall detection accuracy (i.e., the proportion of samples correctly classified, either as benign or as malicious). In addition, we discuss Youden's J statistic to find the optimal trade-off between false positives and false negatives.

3.2.2.1 Training and Classification

First, we randomly extracted 3,500 unique JavaScript files from the email dataset (malicious) and as many from the Microsoft dataset (benign) to build a balanced model. We considered that the remaining samples were unknown, and we leveraged them to measure the detection performance of JAST. To limit statistical effects from randomized datasets, we repeated this procedure five times and averaged the detection results. As indicated in Table 3.3, JAST is able to correctly classify 99.48% of our benign dataset while still detecting 99.46% of the malicious email samples. As both these benign and malicious files are, for the most part, obfuscated, this demonstrates the resilience of our system to this specific form of evasion. More importantly, it shows that JAST does not confuse obfuscation with maliciousness nor plain text with benign inputs. JAST can indeed leverage differences between benign and malicious obfuscation at a syntactic level to distinguish benign obfuscated from malicious obfuscated files. In fact, while the former is used to protect code privacy and intellectual property, the latter aims at hiding its malicious purpose without specific regard to performance. Furthermore, our system offers a very high true-negative rate for the web frameworks, the source code of JavaScript games, and Atom, even though these sample families were not present in the training set. The possible transfer of an email-based model to detect web samples is further discussed in Section 3.3.2.

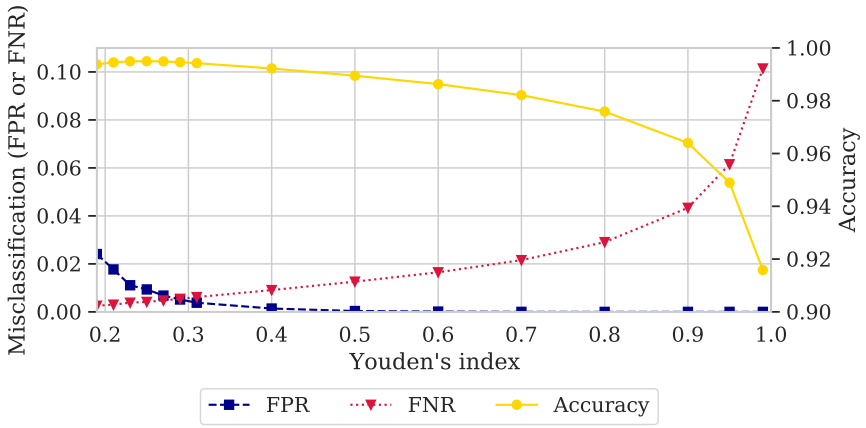


Figure 3.3: Detection performance of JaSt depending on Youden's index

3.2.2.2 Youden's Index

Both the false-positive (0.52%) and the false-negative (0.54%) rates are very low for JaSt. This indicates that, based on a frequency analysis of their 4-grams, our classifier is able to make an accurate distinction between benign and malicious samples almost 99.5% of the time. We achieve this optimal trade-off between false positives and false negatives by using Youden's J statistic [168, 232] where J is defined as:

$$J = \text{sensitivity} + \text{specificity} - 1 = TPR^1 - FPR^2$$

This index corresponds to the area beneath the ROC (Receiver Operating Characteristics) curve [61] subtended by a single operating point. Its maximum value is used as a criterion for selecting the optimal cut-off point between false positives and false negatives. In our case, we determined Youden's index with 5-fold cross-validation on the independent dataset presented in Section 3.1.3. We selected the value 0.29, which means that we consider a sample to be malicious if the probability of it being malicious is above 0.29, according to our random forest classifier (cf. Section 2.3.1). Figure 3.3 presents the evolution of the detection performance when the value of Youden's index varies between 0.19 and 0.99. We obtained the best trade-off between false-positive and false-negative rates with a threshold of 0.29 and the maximum detection accuracy (of 99.5%) with 0.25. Also, with an index of 0.23, we have a reduction of the sharp false-positive rate decline while retaining a low false-negative rate and an extremely high detection accuracy of 99.49%. Besides, trading an extremely low false-positive rate for a higher false-negative rate downgrades the overall detection accuracy extremely rapidly. In the remaining of this chapter, we consider a threshold of 0.25, unless stated otherwise.

[1] True-positive rate
[2] False-positive rate

Tool	FPR	FNR	Static	Dynamic
JaSt	5.2E-3	5.4E-3	✓	-
Cujo	2.0E-5	5.6E-2	✓	✓
PJScan	1.6E-1	1.5E-1	✓	-
Zozzle	3.1E-6	9.2E-2	✓	✓

Table 3.4: Comparison of our detection performance with related work

Youden's index	FPR	FNR
0.7	1.19E-5	2.16E-2
0.8	0	2.91E-2
0.9	0	4.34E-2

(a) On our datasets

Youden's index	FPR	FNR
0.7	1.26E-4	-
0.8	1.68E-5	-
0.9	6.71E-6	-

(b) On samples from Alexa top 10k

Table 3.5: Detection performance of JaSt for several Youden's indexes

3.2.3 Comparison with Related Work

In 2010, Rieck et al. introduced Cujo to detect malicious JavaScript embedded in web pages [173]. For this purpose, they use an SVM classifier, which leverages n-gram features from lexical units. Similarly, in 2011, Laskov et al. developed PJScan, which combines n-grams built upon lexical features with a model of normality, to detect malicious JavaScript embedded in PDF documents [117]. With Zozzle, Curtsinger et al. combined in 2011 features extracted from the AST with their corresponding node value to train a Bayesian classification system to detect malicious JavaScript [50].

As shown in Table 3.4, JaSt is heavily optimized to detect malicious JavaScript instances with its low false-negative rate of 0.54% (threshold 0.29), which is between 10 and 28 times lower than the other tools proposed thus far. Even though Cujo and Zozzle also used the results of a dynamic analysis to detect malicious JavaScript samples more accurately, JaSt has a higher overall detection accuracy. Like the majority of anti-virus systems, they rather traded a low false-positive rate for a higher false-negative rate. In fact, as indicated by Curtsinger et al., based on the number of URLs on the web, a false-positive rate of 5% is considered acceptable for static analysis tools but rates even 100 times lower are not acceptable for in-browser detection.

As mentioned in Section 3.2.2.2, we can leverage Youden's index to shift the false-positive and false-negative rates, according to the system use case and dataset. Therefore, to perform further comparisons with Cujo and Zozzle, we increased the value of Youden's index to lower our false-positive rate. With a threshold of 0.7, our system already has a lower false-positive rate than Cujo while retaining a lower false-negative rate (see Table 3.5a). To ensure that these results were not coming from a lack of benign JavaScript samples, we extracted 119,233 unique benign JavaScript files from Alexa top 10k websites and classified them (see Table 3.5b). Our model does not have such a low false-positive rate on these samples as previously since we are using an email-based model to classify web-JavaScript (this concept is further discussed in Section 3.3.2).

Nevertheless, with a threshold of 0.8, the false-positive rate of JaSt on the Alexa dataset is lower than Cujo while still retaining a lower false-negative rate. As for Zozzle, a threshold of 0.8 on our dataset provides both a better false-positive and a better false-negative rate. As previously, we also performed this comparison on the samples extracted from Alexa top 10k. With a threshold of 0.9, we have a false-positive rate of 6.71E-4% (standing for 0.8 false positives, averaged over five runs)–admittedly a little superior to Zozzle–but still a lower false-negative rate.

Added value of JaSt Several parameters are responsible for the higher detection accuracy of JaSt compared to Cujo and Zozzle. First, we do not trade a very low false-positive rate for a higher false-negative rate. This way, our system accurately detects benign samples with an accuracy of 99.48% while still flagging 99.46% of the malicious ones. As indicated in Figure 3.3, an extremely low false-positive rate significantly decreases the classifier's accuracy. Besides, maximizing the detection accuracy also corresponds to the best trade-off between false positives and false negatives. Furthermore, the choice of our random forest classifier has an impact on the detection performance. It performed indeed better than Bernoulli naive Bayes and SVM, used by Zozzle and Cujo, on our dataset. Last but not least, our syntactic analysis also has an impact on the detection accuracy. For example, Cujo is based on a lexical analysis, which does not perform as well as an AST-based one because lexical units lack context information. We discuss and justify the added value of an AST-based approach, compared to a lexical one, more thoroughly in Chapter 4.

3.3 Applying JaSt in the Wild

We underlined previously both the high true-positive and true-negative rates of JaSt, which therefore outperforms previous work. Based on the accuracy of our system, we next study the temporal evolution of JavaScript samples over a year. Subsequently, we discuss similarities between JavaScript files from different families and showcase that we can leverage an email-based model to classify web samples. We finally report on JaSt run-time performance.

3.3.1 Malicious JavaScript Evolution over Time

In this section, we focus on the temporal evolution of malicious JavaScript extracted from emails received between January 2017 and January 2018. For each month, we perform the two following steps:

- We first consider that the ground truth of the JavaScript samples collected on a given month is unknown. For this purpose, we leverage the model we built in the previous months (the first model being created in January 2017) to classify the JavaScript instances of the considered month;
- In practice, we know the ground truth of the JavaScript samples collected on the considered month (cf. Section 3.2.1). Thus, we leverage these samples to build a new model, including all samples from the previous months.

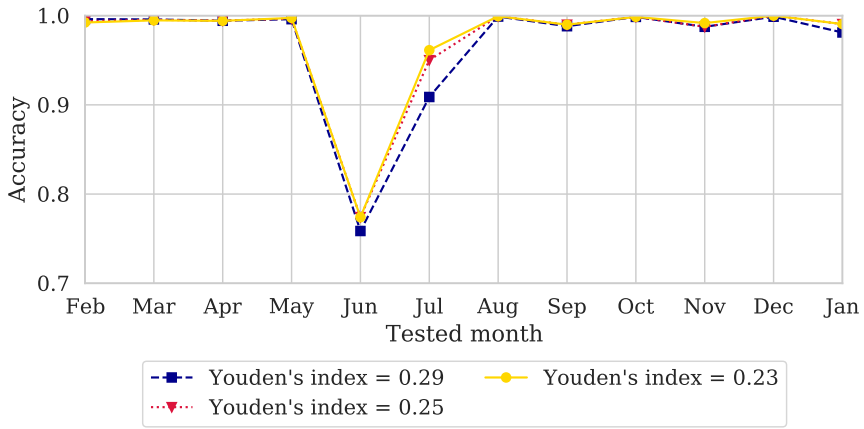


Figure 3.4: Evolution of JAST's accuracy with retraining over time

As a consequence, we classify the samples from January 2018 with a model initially built in January 2017 and extended each month until December 2017 (included), with new and up-to-date malicious as well as benign JavaScript instances. Figure 3.4 shows the detection accuracy of our random forest classifier for three different Youden's indexes. Except for June and July, the detection accuracy stays relatively constant over time, with an average of 96.36% (99.63% without these two months). The predictions decline in June and, to a lesser extent, in July only depends on malicious JavaScript misclassifications (over the whole period, we have a mean false-positive rate of 0.22%). As expected, the decrease gets more important when the value of Youden's index increases, as this threshold represents the probability cut-off to consider a sample as malicious. As a comparison, we replaced the complete relearning of a model each month by an update function adding 100 new trees to the forest built in the previous months. As both experiments presented the same decline in June and July, we performed an in-depth study of these specific samples.

In particular, we combined JSINSPECT, a project built upon the AST to detect structurally similar code [96], with a manual analysis. We then showcased that the misclassifications were coming from several JavaScript waves, each wave containing samples with the same AST. In fact, attackers abused obfuscation techniques to send a unique copy of malicious JavaScript email attachments to each recipient. In this specific case, they only randomized the function and variable names for each JavaScript file they produced. Since their SHA1 hash is different, we did not consider them as duplicate, even though they are identical at the AST level.[3] In June, there are four such misclassified waves with respectively 213, 355, 578, and 1,049 files in them. If one sample of a wave is misclassified, so is the entire wave, which yielded, in our case, a high number of false negatives. We observe a similar phenomenon in July, with two waves containing 107 and 354 misclassified samples. These six specific waves are admittedly composed of malicious samples only, but the classifier labeled each of them as benign with a probability over 78%. On the contrary, we could have observed the

[3]Variable and function names are not part of the AST but represented with an `Identifier` node

Month	#Malicious	#Malicious big waves	Part of a big wave (%)	Part of a FN big wave (%)
Feb	4,894	8	66.92	0
Mar	4,838	4	28.00	0
Apr	4,883	4	30.04	0
May	4,922	4	44.64	0
Jun	4,987	6	73.73	39.74
Jul	4,831	6	53.88	7.32
Aug	6,536	6	64.35	0
Sep	592	0	0	0
Oct	3,610	1	8.80	0
Nov	120	0	0	0
Dec	419	0	0	0
Jan	53	0	0	0

Table 3.6: Insights into our malicious samples

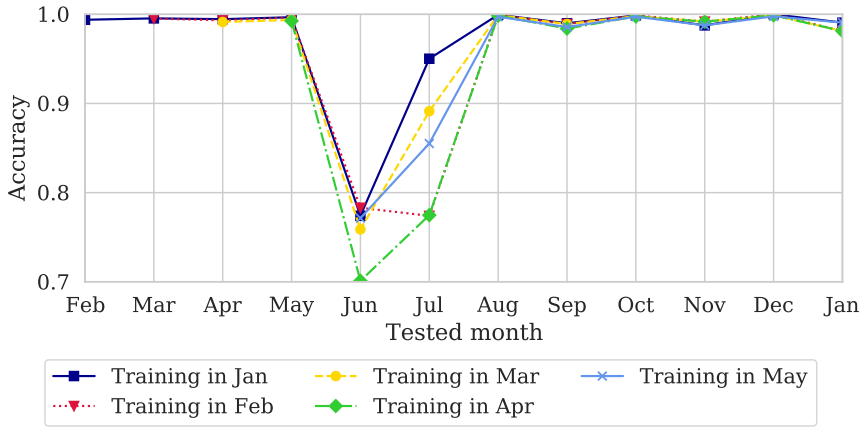


Figure 3.5: Evolution of JAST's accuracy depending on the training month

inverse phenomenon, where one sample of a malicious wave would have been recognized as malicious; thus, as a wave only contains samples with the same AST, the whole wave would have been classified as malicious, yielding, in turn, a high number of true positives. Table 3.6 indicates that this is generally not the case, as we reported the biggest malicious waves[4] rather in June and July. Besides, there are only big wave misclassifications in June and July, as the other waves are being correctly flagged as malicious with a probability over 50% (in general over 75%).

As a second experiment, Figure 3.5 presents the evolution of the detection accuracy when the month used to build the initial model varies (the rest of the experiment stayed unchanged). As previously, the accuracy stays relatively constant except for the decline in June and July, for the reasons mentioned before. In particular, the accuracy evolution in June between a model first built in April and the other models highlights the presence of these big waves (e.g., more misclassifications with the April model), with a similar observation in July.

[4]We define a *big wave* as a wave containing more than 300 syntactically similar samples

This way, to be long-time effective, JaSt has to adapt to new JavaScript instances. This adaptation process is achieved by extending the set of n-gram features used with new and up-to-date JavaScript samples. In fact, our analysis of JavaScript instances over a year has shown that building a model each month to detect malicious variants in the current month is only effective if the training set contains enough files, which are representative of the distribution found in the wild and is not a compilation of different JavaScript waves received in the past few months.

3.3.2 Analysis of Web-JavaScript

In Section 3.2.2, we trained JaSt with malicious JavaScript extracted from emails (and Microsoft samples for the benign part). Still, this model yielded accurate predictions for web frameworks, the source code of Atom, and for JavaScript games, even though these families were not represented in our training set. For this reason, we discuss the applicability of an email-based model to detect other types of JavaScript, such as web samples. In particular, we focus on malicious HTML email attachments, exploit kits, and Alexa top 10k websites. We finally discuss our choice to separate the email-JavaScript from the web-JavaScript analysis and highlight syntactic similarities between different malicious JavaScript families.

3.3.2.1 Experimental Protocol

In this experiment, we used the five previously constructed email-based models to classify web samples. In particular, we collected inline JavaScript extracted from malicious HTML email attachments (provided by the BSI), exploit kits (EK) from 2010 to 2017 (provided by Kafeine DNC [100]), and Alexa top 10k websites. Regarding Alexa, we also extracted third-party scripts and considered that all scripts were benign. Given that we extracted JavaScript from the start pages of the ten thousand websites with the highest ranking, we assume these samples to be benign. Although it has been shown that these websites could host malicious advertisements, our JavaScript extraction process, which relies on statically parsing the web page with Python and extracting `script` and `src` tags, protected us from these elements generated dynamically. Figure 3.6 presents the detection accuracy (in terms of true-positive rate for the malicious files and true-negative rate for Alexa) on these web samples. In particular, we consider that an HTML page or an exploit kit is benign if all the contained JavaScript snippets are classified as benign. If one malicious JavaScript sample was detected, we labeled the whole page/exploit kit as malicious.

3.3.2.2 Analysis of Malicious Web-JavaScript

JaSt was able to detect 82.31% of the 13,595 malicious web-JavaScript inputs, which underlines some similarities at the 4-gram level between email-JavaScript and web-JavaScript. Further insights into the false negatives indicate that 14.37% of the samples have been *correctly* classified as benign. In fact, a manual inspection of 80 exploit kits showed that in 21.25% of the cases, the malicious part was *not* embedded in JavaScript samples. Instead, the attack vector was either contained in an SWF bundle, or the

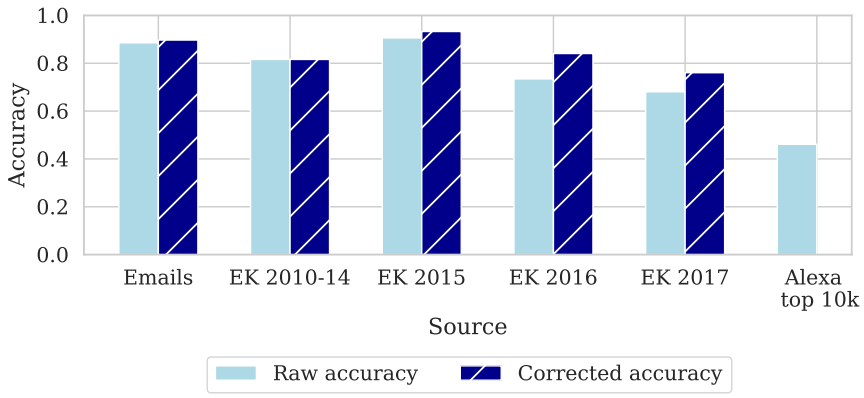


Figure 3.6: Accuracy of JaSt on web-JavaScript after email-based training

exploit kit merely included a resource trying to exploit an existing flaw without any scripting code at all. Another issue was related to the quality of JavaScript samples: while analyzing 110 malicious email attachments and exploit kits, we discovered that some files were broken and could, therefore, not be parsed. In 3.64% of the cases, the *malicious* part could not be parsed, therefore could not be analyzed. This naturally means that, in an attack, this code would not have been executed. As a consequence, when considering only HTML documents and exploit kits, which could be entirely parsed and whose malicious behavior was included in a JavaScript snippet, we got a corrected true-positive rate of 85.18%, which represents an improvement of 3.36%.

While malicious email-based and web samples present some similarities, they also have syntactic differences, which prevented JaSt to provide as much confidence in detecting malicious web-JavaScript as in email-JavaScript. In fact, web samples tend to contain less malicious patterns, have comments at regular intervals, and benign snippets next to the malicious parts. Also, they do not use the same obfuscation techniques as malicious email-JavaScript. As malicious email-JavaScript aims at providing a unique copy for each recipient, it abuses variable and function names randomization, data obfuscation, and encoding obfuscation. On the contrary, malicious web-JavaScript rather tends to identify software vulnerabilities in clients' machines and exploit them to upload and executes malicious code on the client side. For this purpose, the attackers preferably use variable and function names randomization and neatly package their code, which can slightly degrade JaSt accuracy.

3.3.2.3 Analysis of Benign Web-JavaScript

Next, we focus on detecting benign JavaScript samples extracted from Alexa top 10k websites. JaSt could detect 46.11% of them. Instead of grouping all JavaScript snippets of a web page and labeling the website as benign if all samples were recognized as benign, we performed a second experiment. We collected all JavaScript snippets from Alexa top 10k (between 100 bytes and 1 megabyte), and we analyzed them independently. In total, we extracted 119,125 JavaScript samples and got a true-negative rate of 92.79% (averaged over five runs); thus, a false-positive rate of 7.21%, which is, this time, more

	Parsing	N-gram analysis	Training	Update	Classification	Sum
Time (s)	96.55	14.43	1.79	0.41	0.26	113.44
Time (%)	85.12	12.72	1.57	0.36	0.23	100

Table 3.7: Run-time performance of JAST on 500 samples

in line with our results from Section 3.2.2. If we consider that an Alexa top 10k website contains n JavaScript snippets, we can expect a true-negative rate of 0.9279^n. On average, it contains 16 snippets; therefore, the probability of having a true negative is 30.18%, which is lower than our true-negative rate of 46.11% and underlines the artificial performance decrease we would introduce by classifying an HTML document instead of a JavaScript code snippet.

3.3.2.4 Discussion

We chose not to include any JavaScript extracted from HTML documents for the training and evaluation part of this chapter (i.e., Section 3.2.2) but rather to discuss the extension of an email-based model to detect web samples for three reasons. First, we did not have any ground truth regarding the position of the malicious entity in malicious HTML documents, which would have required a systematic analysis of our 13,595 snippets to detect and use only the *malicious* JavaScript samples to train our classifier with. For this reason, we decided to exclude them from the evaluation and instead chose to flag any HTML documents containing at least one malicious JavaScript snippet as malicious. For symmetry purposes, we applied the same procedure to benign HTML files, which thereby reduced the number of benign scripts in our dataset. Last but not least, splitting email and web evaluation was a way to show that syntax-based features can be core in classifying JavaScript instances: no matter the obfuscation used to hide their functionality, malicious JavaScript samples do not necessarily hide their true function.

3.3.3 Run-Time Performance

Finally, we evaluated the run-time performance of JAST on a commodity PC with a quad-core Intel(R) Core(TM) i3-2120 CPU at 3.30 GHz and 8 GB of RAM. Table 3.7 shows the processing time for all stages of our method on 500 unique JavaScript samples (half of which are benign, the other half being malicious), which we randomly selected. The most time-consuming operation is the parsing with Esprima (written in JavaScript and called from our Python program), which accounts for over 85% of the overall detection time. In comparison, the production of all 4-grams and the creation of a 2,457-dimension frequency vector is relatively fast (12.72% of the time). Finally, building or updating a model and classifying JavaScript inputs represents, on average, less than 1% of the processing time. Overall, JAST classified 500 JavaScript files in less than 2 minutes. Compared to PJSCAN (implemented in C with its own C library to classify JavaScript entities), which can analyze a PDF document in 0.0032 seconds, our approach is slower. In compensation, the accuracy of our predictions is significantly

better (cf. Section 3.2.3). Similarly, JaSt can analyze on average 0.14 MB/s, which is comparable to the 0.2 MB/s of Zozzle for the same amount of features, and we also retain a higher detection accuracy (99.46% vs. 99.20%). We naturally envision that JaSt would be parallelized for a deployment in the wild.

3.4 Summary

Many malicious JavaScript samples today are obfuscated to hinder the analysis and the creation of signatures. To countermand this, and due to the large volume of JavaScript files in the wild, we proposed our static analyzer JaSt to quickly process the vast majority of samples. Specifically, our system combines an extraction of features from the AST with a random forest classifier to detect malicious JavaScript instances at scale (*RQ1*). Due to our usage of AST-based patterns, our approach can handle obfuscated inputs and does not confuse obfuscation with maliciousness. In practice, JaSt yields both a very high true-positive (99.46%) and true-negative rates (99.48%) and outperforms previous detectors. In addition, we further discussed the evolution of our accuracy over time and highlighted some syntactic similarities between different classes of malicious JavaScript.

Even though JaSt has a very high detection accuracy, it is not infallible and still leads to misclassifications. In fact, it lacks semantic information to go beyond solely relying on the code syntax. In the following chapter, we present JStap, which goes beyond the code structure by also considering control and data flows to detect malicious JavaScript inputs.

4

JSTAP: A Static Pre-Filter for Malicious JavaScript Detection

In the previous chapter, we presented JaSt, our static approach, which leverages the AST to detect malicious JavaScript instances. While JaSt has an overall detection accuracy of almost 99.5%, it still misclassifies some inputs. The same applies to prior work, such as the lexical detector Cujo [173] and the AST-based Zozzle [50]. These systems even misclassify different samples. In fact, these static detectors solely rely on the code structure to detect malicious JavaScript inputs; thus, they lack semantic information to go beyond the lexical and syntactic order of the code.

In this chapter, we focus on *RQ2: Can we add more semantic information into the AST of JavaScript files? Specifically, to what extent can we statically enhance the AST with control and data flows? Which features, combined with machine learning algorithms, work best to detect malicious JavaScript instances?* For this purpose, we propose JStap, our static detector, which is composed of ten modules, including five ways of abstracting code, with differing levels of context and semantic information, and two ways of extracting features. Similarly to JaSt, we train a random forest classifier for each module. In practice, JStap outperforms existing systems, which we reimplemented and tested on our dataset totaling over 273,216 samples. To further improve the detection accuracy of our approach, we also combine the predictions of several modules so that only samples with conflicting labels would need further scrutiny.

4.1 Presentation of our Modular Detector

JStap, which we implemented in Python, is composed of several modules, which can run, independently or combined, to accurately detect malicious JavaScript instances. The architecture of each module consists of an abstract code representation (Figure 4.1 stage 1), a feature-extractor (stage 2), and learning components (stage 3). First, we perform a static analysis of JavaScript samples, leveraging the AST to build the Control Flow Graph (CFG) and the Program Dependence Graph (PDG) (Section 4.1.1). We adopt a definition of the CFG and PDG that slightly differ from Allen [3] and Ferrante et al. [64], as we chose to enhance the AST with control and data flow edges, for the reasons mentioned in Section 2.2. Subsequently, we traverse the resulting graphs by following the control and/or data flow edges to extract syntactic units, whose combination still carries the original control and/or data flow semantics. We also consider lexical units

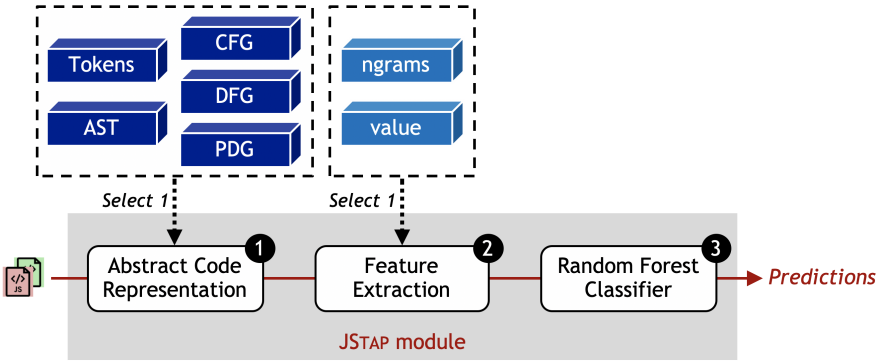


Figure 4.1: Architecture of JStap with a focus on one module

Token	Value	Token	Value	Token	Value
Identifier	x	Numeric	1	Punctuator	)
Punctuator	.	Punctuator	;	Punctuator	{
Keyword	if	Keyword	if	Identifier	d
Punctuator	=	Punctuator	(	Punctuator	=
Numeric	1	Identifier	x	Identifier	y
Punctuator	;	Punctuator	.	Punctuator	;
Keyword	var	Keyword	if	Punctuator	}
Identifier	y	Punctuator	==		
Punctuator	=	Numeric	1		

Table 4.1: Lexical units (tokens) extracted from Listing 4.1

and syntactic units extracted from the AST to extend our approach with node context information, since control and data flow edges only link statement nodes together. In particular, we combine the extracted units by groups of n to build n-gram features. At the same time, and independently of the prior approach, we also combine these units with variable name information (Section 4.1.2). In both cases, we use the frequency of the extracted features as input to our random forest classifier, which learns to distinguish benign from malicious JavaScript samples (Section 4.1.3). In the following sections, we discuss the details of each stage in turn.

4.1.1 Abstract Code Representations

We chose to perform a static analysis to detect malicious JavaScript instances due to its speed, precision, and code coverage. In particular, we can leverage different levels of code abstraction, with more or less semantic information, to identify recurrent programmatic and structural constructs specific to malicious or to benign inputs. We focus on four different abstraction levels, which we introduced in Section 2.2.[1] First, we consider a lexical analysis, which directly processes the code, one word after the other. On the contrary, an AST-based analysis takes into account the JavaScript grammar; thereby, the syntactic structure of the program. Next, the CFG contains semantic information, compared to the AST, as it considers the conditions that have to be met for a specific execution path to be taken. Finally, the PDG also takes into account the dependencies between variables. This way, with these code representations, we can process JavaScript inputs at different static levels. This naturally means that we can combine these representations to model the different code properties more accurately.

4.1.1.1 Lexical Unit Extraction

First, we perform a lexical analysis of JavaScript files with the tokenizer Esprima [79]. Specifically, it linearly converts the source code into a list of abstract symbols representing lexical units (e.g., `Keyword`, `Identifier`). Still, this technique neither uses the context in which a given word appears nor the overall syntactic structure of the snippet

[1] Ultimately leading to five ways of abstracting the code, cf. Section 4.1.2.1

```
1 x.if = 1;
2 var y = 1;
3 if (x.if == 1) {d = y;}
```

Listing 4.1: JavaScript code example

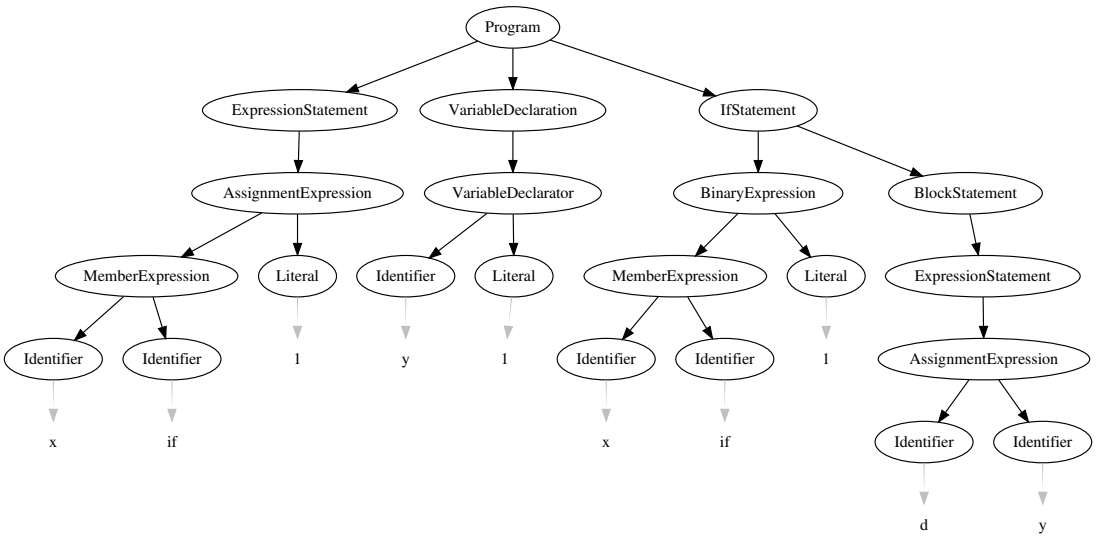


Figure 4.2: AST of Listing 4.1

it analyses. As an illustration, Table 4.1 presents the tokens extracted from the code snippet of Listing 4.1. In particular, for the assignment *x.if = 1*, a lexical analysis is unable to infer that the traditionally reserved word *if* is not used as a `Keyword`, but as an `Identifier` (see Table 4.1 line 3).

4.1.1.2 AST Generation

Contrary to the token approach, the AST describes the syntactic structure of an input sample, as it rests upon the JavaScript grammar [56]. In particular, we generate the AST with the parser Esprima, which can produce up to 69 different syntactic units, referred to as nodes. Inner nodes represent *operators*, such as `VariableDeclaration`, `AssignmentExpression`, or `IfStatement`, while the leaf nodes are *operands*, e.g., `Identifier` or `Literal` (except for `ContinueStatement` and `BreakStatement`). As an illustration, Figure 4.2 shows the Esprima AST obtained from the code snippet of Listing 4.1.[2] This time, the construct *x.if* is recognized as a `MemberExpression` with *x* and *if* being both correctly labeled as `Identifier` units. Still, the AST only retains information about the nesting of programming constructs to form the source code but does not contain any semantic information such as control or data flow.

[2]For legibility reasons, the variable names and values appear in the graphical representations of the AST, CFG, and PDG but they are not part of the graphs

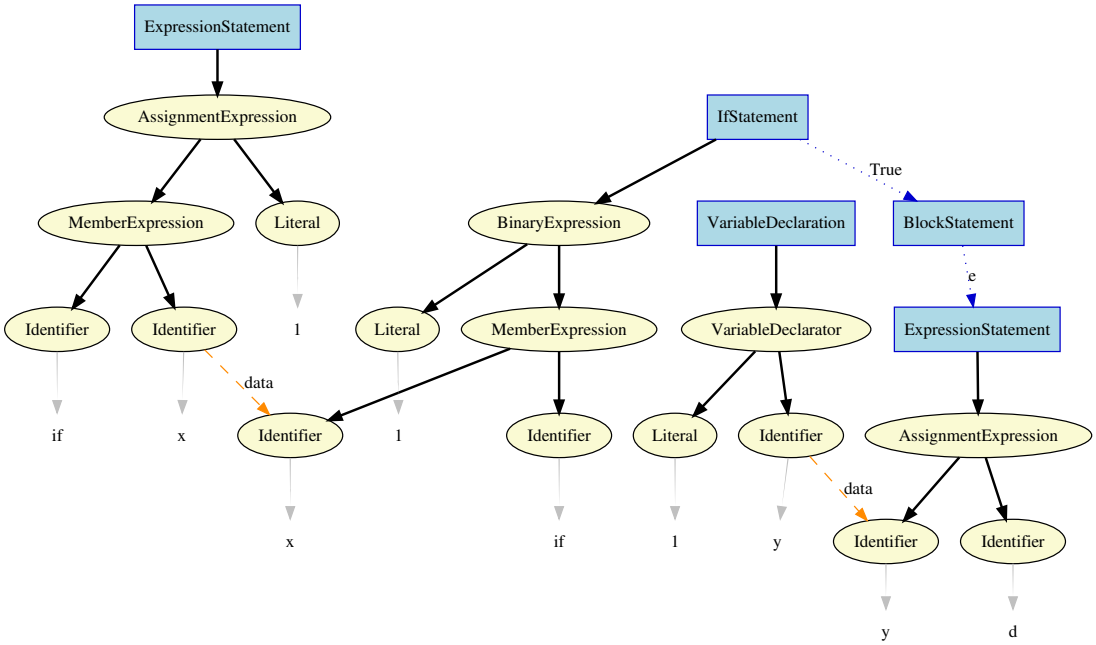


Figure 4.3: AST of Listing 4.1 extended with control & data flows

4.1.1.3 CFG: AST + Control Flow

Subsequently, JSTAP extends the AST with control flow to reason about the conditions that should be met for a specific execution path to be taken. We refer to the resulting structure as the CFG (Control Flow Graph). We construct the CFG by traversing the AST nodes depth-first pre-order. In particular, we represent flows of control on statement nodes, which we connect with labeled and directed edges. We label edges originating from predicates with a boolean, standing for the value the predicate should evaluate to, for its descendants in the graph to be executed. For non-predicate statement nodes, we use the label `e`, standing for epsilon control flows. In addition to the nodes defined as *statement* by the JavaScript grammar [56], we also consider `CatchClause` and `ConditionalExpression`, as they both have an impact on the program execution flow. Contrary to the AST of Figure 4.2, Figure 4.3 (considering only the blue dotted control flow edges) shows an execution path difference when the *if* condition is true, and when it is not. Still, the CFG does not enable to reason about variable dependencies nor the order in which the resulting statements are executed.

4.1.1.4 PDG: AST + Control Flow + Data Flow

Finally, JSTAP adds data flow information to the CFG, which we refer to as a PDG. In particular, we connect two statement nodes with a directed data flow edge if and only if they have a variable child (also including object and function) defined or modified at the source node and used at the destination node, taking into account its reaching definition.

In JavaScript, a scope defines the accessibility of variables. If a variable is defined outside of any function or without the `var`, `let`, or `const` keywords, or using the

`window` object, it is in the global scope. On the contrary, variables that can be used only in a specific part of the code, e.g., block statement, are in a local scope. To build our PDG, we traverse the CFG depth-first pre-order and maintain two lists of variables. The first one contains the global variables and the second one the local variables currently declared in the *considered block statement*, taking into account the specific local scope of variables defined with the *let* or *const* keywords (in the block where they are defined). For objects, we keep the order in which they are modified (e.g., whenever a method is called on them or one of their properties changes). In fact, we cannot always statically predict which method should be called on an object first, e.g., an `XMLHttpRequest` must be opened before the `send()` method is called. In such cases, we add a data flow edge between the previous version of an object and the current one, and we update our variable list (local or global according to the context) with a reference to the modified object. In addition, JStap can analyze functions, respects their scoping rules, and handles closures and lexical scoping. In particular, we handle function names as variables (local or global), since functions and variables cannot share a name in JavaScript. Also, we make the distinction between a function declaration–i.e., a standalone construct defining a named function variable– and a function expression–i.e., a named or anonymous function that is part of a larger expression. Finally, we connect function call nodes to the corresponding definition nodes with data flow edges, thus defining the PDG at program level [229].

Overall, this code representation captures the data and control flows between the different program components. Therefore, it is not influenced by arbitrary sequencing choices made by programmers but represents the semantic order of the code. Contrary to the AST of Figure 4.2, Figure 4.3 indicates the order in which statements from Listing 4.1 should be executed.[3] For example, as shown by the data flows (orange dashed edges), lines 1 and 2 are executed before line 3; we could nevertheless swap lines 1 and 2 without altering the code semantics. In the remaining of this thesis, we graphically represent the flows of control with blue dotted edges and data flows with orange dashed edges. The remaining AST edges (i.e., that do not already have a control flow edge) are displayed in bold and the variable names and values (that do not belong to the graphs) with gray tapered edges. Also, we represent statement nodes in blue squares, while non-statement nodes are in yellow circles.

4.1.2 Feature Extraction

Once JStap built abstract code representations to analyze JavaScript samples, we extract lexical units, and we traverse the different tree/graphs to collect syntactic units (Section 4.1.2.1). Overall, we consider five different ways of abstracting the code. Subsequently, and for each code representation, we consider (independently) n-gram features and the combination of the extracted units with their corresponding node value (e.g., variable name), hence two ways of extracting features, meaning that JStap is composed of ten modules (Section 4.1.2.2). Finally, for each module independently,

[3]For legibility reasons, we draw the data flow edges between leaf nodes instead of their corresponding nearest statement nodes

learning components take the frequency of the extracted features as input for the classification process (Section 4.1.3).

4.1.2.1 Graph Traversals

As far as the lexical analysis is concerned, we already extracted lexical units (tokens) in Section 4.1.1.1. For the AST, CFG, and PDG, we traverse each graph by following its specific edges to extract the name of each node, referred to as a syntactic unit. Specifically, for the AST, a depth-first pre-order traversal of Figure 4.2 gives the following syntactic units: `ExpressionStatement`, `AssignmentExpression`, `MemberExpression`, [...] `Identifier` (the `Program` node just represents the root of the AST and does not have any syntactic meaning). For the CFG, we also traverse the AST but only store nodes linked by a control flow edge (i.e., the *e*, *True*, and *False* labels), e.g., in Figure 4.3: `IfStatement`, `BlockStatement`, and `ExpressionStatement`. In practice, considering only statement nodes is not informative enough to distinguish benign from malicious JavaScript inputs (due to both of them linking the same statements with one another, cf. Section 4.2.2.1). To add more context information to the control flow-based units, we also traverse once the sub-AST of each node having a control flow edge. For example, in Figure 4.3, JSTAP reports the `IfStatement` node and traverses its sub-AST before following the control flow edges and traversing the `BlockStatement`, then the `ExpressionStatement` nodes. This time, we do not traverse their corresponding sub-ASTs, as we already performed this step while handling the `IfStatement` node.[4] Finally, the process is similar for the PDG, with consideration of the data flow edges. In the following, we use the term *DFG* (Data Flow Graph) to refer to the PDG traversal only along data flow edges and the term *PDG* to refer to the PDG traversal along the data flow edges, followed by a second traversal along the control flow edges.

4.1.2.2 Feature Analysis

For the five abstract code representations, namely tokens, AST, CFG, DFG, and PDG, we (independently) build features by considering n-gram patterns and the combination of the extracted units with their corresponding node value. Therefore, JSTAP contains ten modules, with five different static code analysis levels and two ways of representing features extracted from these different code representations.

N-Gram Features To identify specific patterns in JavaScript documents, in the first scenario, we move a fixed-length window of n symbols over the list of lexical or syntactic units previously extracted to get every sub-sequence of length n (n-grams) at each position. For example, the three first 2-grams of Table 4.1 are: `(Identifier, Punctuator)`, `(Punctuator, Keyword)`, and `(Keyword, Punctuator)`. As mentioned in the previous chapter, n-grams are an effective means for abstracting

[4] At the end, we retain the following units: `IfStatement, BinaryExpression, MemberExpression, Identifier, Identifier, Literal, BlockStatement, ExpressionStatement, AssignmentExpression, Identifier, Identifier, BlockStatement, ExpressionStatement`

	Tokens	AST	CFG	DFG	PDG
ngrams	602	11,050	18,105	17,997	24,706
value	24,912	45,159	36,961	45,566	46,375

Table 4.2: Number of selected features per module

the code. We empirically evaluated different n values, and we selected $n = 4$, which provides the best trade-off between detection accuracy and run-time performance. In the following, we use the term `ngrams` to refer to the 4-gram features we built as described above.

Node Value Features In the second scenario, we do not use n-gram features, but we combine each lexical unit with its corresponding value (as presented in Table 4.1) and each syntactic unit extracted from the AST, CFG, DFG, or PDG with its corresponding `Identifier/Literal` value. For example, the first two features of the AST from Figure 4.2 are (`ExpressionStatement`, x) and (`AssignmentExpression`, x). In the following, we use the term `value` to refer to the features combining lexical or syntactic units with their corresponding values, as described above.

4.1.2.3 Feature Space

Next, we leverage the frequency of these `ngrams` or `value` features to determine if a given input is either benign or malicious. In fact, as mentioned in the previous chapter, JavaScript samples sharing several features with the same frequency present similarities with one another, while files with different features have a more dissimilar content.

To compare the frequency of the features appearing in several JavaScript files, we construct a vector space such that each feature is associated with one consistent dimension, and its corresponding frequency is stored at this position in the vector. To limit the size of the vector space, which directly impacts the performance, we use the χ^2 test to check for correlation. We select only features for which $\chi^2 \geq 6.63$, meaning that feature presence and script classification are correlated with a confidence of 99% [223]. Table 4.2 presents the number of features considered for each of the ten JStap modules based on our training set, which we describe in Section 4.2.1. For the `ngrams` variant, there are more statistically representative features when the complexity of the code representation increases, i.e., complex graph structures lead to more edges. This also holds for the `value` approach, except for the CFG traversal, for which we both have fewer representative features and fewer features in general than for the AST or PDG. We assume that this comes from benign and malicious actors using more similar variable names in statements with a control flow than in other statements. This is confirmed to some extent in Section 4.2.2.2, where this approach does not perform as well as the other ones. Finally, we store the frequency of each feature in Compressed Sparse Row (CSR matrix) [209] to efficiently represent non-zero values.

Source	Collection	#Samples	Obfuscated
BSI	2017-2018	83,361	yes
Hynek	2015-2017	29,558	yes
DNC	2014-2018	12,982	yes
VirusTotal	2018	3,056	yes
GoS	2017	2,491	yes
Sum	2014-2018	131,448	yes

Table 4.3: Malicious JavaScript dataset

4.1.3 Learning and Classification

The learning-based detection completes the design of our modules. We first build and leverage the CSR matrix of a representative and balanced set of both benign and malicious JavaScript files to train our classifiers (one classifier per module). We empirically evaluated several off-the-shelf systems (Bernoulli naive Bayes, multinomial naive Bayes, SVM, and random forest) and selected random forest, which provided again the most reliable detection results, with the best true-positive and true-negative rates.

4.2 Evaluating JSTAP Modules

In this section, we highlight the accuracy of our ten modules. For this purpose, we leverage high-quality datasets from various sources, totaling over 270,000 unique JavaScript samples. In particular, we analyze, compare, and discuss the detection performance of JSTAP modules. Subsequently, we compare them to state-of-the-art detectors, which we reimplemented to evaluate them on our datasets. Finally, we report on the run-time performance of our ten modules.

4.2.1 Experimental Protocol

The experimental evaluation of our approach rests upon two extensive datasets, with a total size over 6.2 GB. The first one contains 131,448 SHA1-unique malicious JavaScript samples and the second one 141,768 unique benign files. Next, we leverage these datasets to train our random forest classifiers.

4.2.1.1 Malicious Dataset

Our malicious dataset (Table 4.3) is a collection of samples mainly provided by the German Federal Office for Information Security (BSI) [24]. These samples have been labeled as malicious based on a score provided by the combination of anti-virus systems, malware scanners, and a dynamic analysis. To reduce possible similarities between samples from the same source, we got the malware collection of Hynek Petrak (Hynek) [86], exploit kits from Kafeine DNC (DNC) [100] and GeeksOnSecurity (GoS) [68] as well as additional samples from VirusTotal [210]. Most of these samples are obfuscated, e.g.,

Source	Collection	#Samples	Obfuscated
Tranco top 10k	2019	122,910	N/A
Microsoft	2015-2018	16,271	yes
Games	N/A	1,992	no
Web frameworks	N/A	427	N/A
Atom	2011-2018	168	no
Sum	-	141,768	-

Table 4.4: Benign JavaScript dataset

string manipulations, dynamic arrays, or multiple encodings. For performance reasons, we limited our analysis to samples with a PDG smaller than 10 MB.

Even though the samples are malicious, in some cases, we extracted JavaScript from HTML documents and thereby had to ensure that the maliciousness laid in the scripts and was not, e.g., contained in an SWF bundle. For this purpose, we manually analyzed our 19,942 JavaScript samples extracted from HTML documents, 15,475 of which were malicious (we discarded the other samples, which are also not represented in Table 4.3). Since our analysis is entirely static, it provides a complete coverage of the available code. In turn, it is unable to consider dynamically generated JavaScript. To this end, we parsed each malicious file with Esprima and automatically inlined all code passed through `eval` (for invocations with static strings). Thereby, we increased the code coverage of JStap on 1,868 unique scripts, as we did not merely consider a `CallExpression` node with a fixed string parameter anymore but the code contained in the string, possibly (depending on JStap selected module) along with its control and/or data flows. Also, 1,094 samples used conditional compilation [143], which Esprima parses as a large comment. Thus, we automatically replaced this construct with the corresponding code for the parser to produce the actual ASTs of such scripts.

Overall, our malicious dataset contains different samples performing various activities. For example, we have JScript-loaders leading to, e.g., drive-by download or ransomware attacks, and exploit kits (e.g., Blackhole, Donxref, or RIG) targeting vulnerabilities in old versions of Java, Adobe Flash, or Adobe Reader plugins, also trying to exploit old browsers versions.

4.2.1.2 Benign Dataset

As for our benign dataset (Table 4.4), we used Chromium to visit the start pages of Tranco top 10k websites [118].[5] For each visited web page, we waited for the load of the page and observed the site for one second, to also collect dynamically generated scrips. In particular, we stored all inline scripts from the same document in one file–keeping the order in which they are executed–and considered all external scripts separately. This way, we obtained 122,910 unique JavaScript files. Given the fact that we extracted JavaScript from the start pages of high-profile websites, we assume our dataset to be

[5]Even though we visited the websites in 2019, the scripts were not necessarily written in 2019 so that our malicious dataset is not older than our benign dataset

benign. Based on a study from Skolka et al. [182], over 30% of first-party scripts are either obfuscated or minified and over 55% of third-party scripts; therefore, we assume our Tranco dataset to be partially obfuscated/minified. In addition, we consider benign JavaScript from Microsoft products, the majority of which are also obfuscated, which enables us to ensure that JSTAP does not confuse obfuscation with maliciousness. As we also own malicious scripts from Microsoft (i.e., JScript-loaders), we are not introducing a bias in our dataset, even if Microsoft uses custom obfuscation methods. Finally, we collected benign JavaScript from open-source games, web frameworks, and Atom [8]. As these samples may contain new or specific coding styles, we show that JSTAP does not mistake unknown or unusual structures for maliciousness either.

4.2.1.3 Classifier Training

Next, to train a random forest classifier per JSTAP module, we randomly selected 10,000 JavaScript samples from our malicious dataset. We deemed our malicious training set to be representative of the distribution found in the wild due to our multiple malware sources and random selection from an initial malicious pool with over 130,000 samples. Similarly, we randomly selected 10,000 benign files to build a balanced model. In the rest of this chapter, we consider that the remaining samples are unknown and use them to evaluate the detection performance of JSTAP's ten modules. In addition, we extracted all features present in our training set before randomly selecting 5,000 new unique malicious and as many benign samples, to check on this validation set which features are correlated with the classification, using the χ^2 test described in Section 4.1.2.3. In the remainder of this chapter, we consider only these features.

We specifically chose to assemble balanced datasets, even though, in reality, benign web pages outnumber malicious ones. With TESSERACT [165], Pendlebury et al. argue that using unrealistic assumptions about the ratio of benign samples to malware in the data can lead to inflated detection results. In our case, it is not an issue, since we specifically chose metrics to evaluate the detection performance of JSTAP both on benign *and* malicious samples, and we do not merely consider a score to rate the proportion of correct predictions of our modules. Finally, to limit any statistical effects from randomized datasets, we repeated the previous procedure five times and averaged the detection results. Contrary to 5-fold cross-validation, we explore more ways of partitioning data, as each sample is not necessarily tested only once.

4.2.2 JSTAP Detection Performance

After training our random forest classifiers, we classified the remaining samples. First, we discuss the detection performance of all JSTAP modules in terms of true-positive and true-negative rates. Specifically, we chose to compare the accuracy of the different modules over these metrics, and not AUC [61] or F-measure [168], to evaluate how well our modules can detect both benign *and* malicious inputs. In fact, AUC and F-measure would be heavily biased by the composition of our test sets (proportion of benign and malicious samples), while we aim at having a more realistic estimation of our modules' accuracy, both on benign and on malicious samples. Subsequently, we

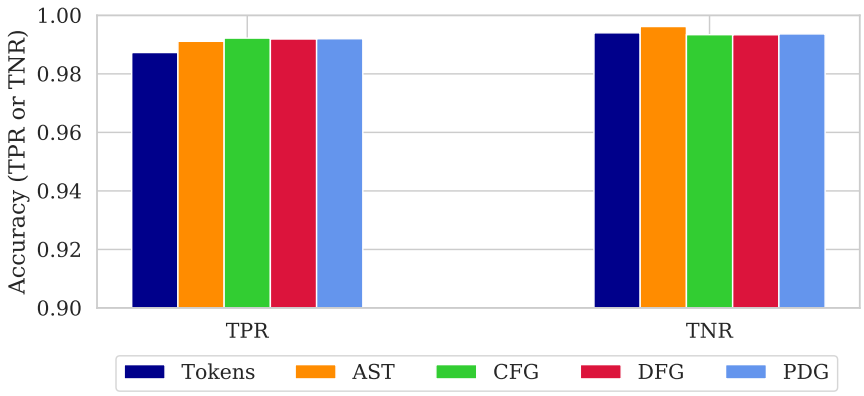


Figure 4.4: Detection performance of JSTAP's `ngrams`-based modules

argue which modules perform best. Finally, we discuss our modules' most important features for classification.

4.2.2.1 `ngrams` Features

In the first scenario, we consider the `ngrams` approach. As Figure 4.4 shows, both the true-positive (TPR) and true-negative rates (TNR) of JSTAP stay constant across our five analyses. Specifically, the TPR ranges from 98.73% (tokens) to 99.22% (CFG), making the CFG the most reliable malicious JavaScript detector in this configuration. As for the TNR, it ranges from 99.34% (PDG) to 99.62% (AST), meaning that the AST detects benign JavaScript best. In terms of overall detection rate, defined as the proportion of samples correctly classified, the AST performs best with an accuracy of 99.38%, whereas the token-based approach performs worst with a detection rate of 99.08%, while CFG, DFG, and PDG have similar detection rates between 99.27% and 99.28%.

As mentioned in Section 4.1.1.1, the lexical level of code abstraction does not use the context (in terms of syntactic structure) in which a given token occurs (e.g., `IfStatement`, `ForStatement`, or `VariableDeclaration`) but merely processes JavaScript inputs one word after the other. For example, the following two JavaScript snippets *for(i = 0; i < 5; i++)* and *if(i == 1) j = 2; k--;* are composed of exactly the same tokens (namely `Keyword`, `Punctuator`, `Identifier`, `Punctuator`, `Numeric`, `Punctuator`, `Identifier`, `Punctuator`, `Numeric`, `Punctuator`, `Identifier`, `Punctuator`, `Punctuator`) but perform different actions. On the contrary, the AST-based analysis leverages the JavaScript grammar, which provides more insights than an analysis purely based on tokens and makes the distinction, e.g., between the `for` and `if` constructs previously highlighted, which leads to a better detection accuracy.

Even though the AST-based approach performs best, the CFG, DFG, and PDG variants are also reliable. Still, to distinguish benign from malicious JavaScript inputs, we observe that the AST code representation may be slightly more informative than the control and data flows. In fact, we ran the same experiments where we extracted the CFG, DFG, and PDG features only by following the control and/or data flow edges,

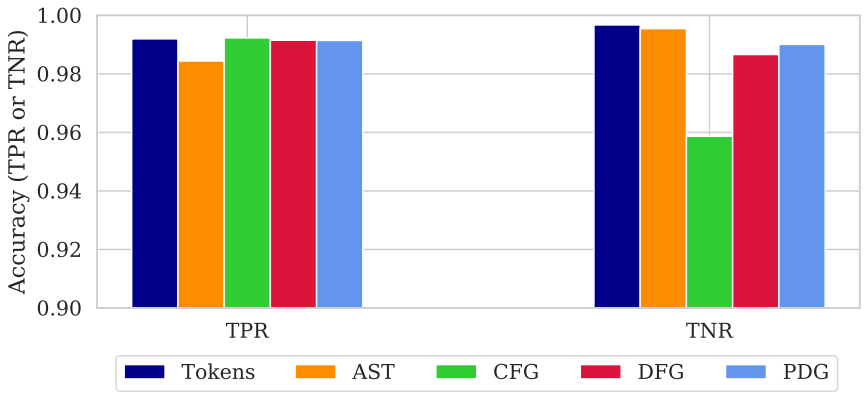


Figure 4.5: Detection performance of JSTAP's `value`-based modules

without also traversing the sub-AST of the corresponding nodes (cf. Section 4.1.2.1). The TPR stayed relatively similar to the results from Figure 4.4, between 98.87% (DFG) and 99.33% (CFG), but the TNR decreased between 94.92% (DFG) and 95.50% (PDG). In fact, the control and data flows are represented only between statement nodes, which are less representative of benign vs. malicious intent than the whole AST structure. Specifically, we extracted the five features most representative of malicious vs. benign intent, for all five `ngrams` modules, according to the corresponding random forest models [178]. For these five modules, `Identifier` nodes are part of the five most important features, which highlights the importance, in terms of prediction accuracy, of not solely relying on statement nodes. Thus, adding AST information into the CFG, DFG, and PDG improved their detection performance up to the AST standards. Still, these three approaches may inherently be limited if there are no control or data flows in the considered files. Out of the 253,216 samples we classified (we excluded the samples from the training set), the CFG could handle on average 231,490.8 of them (91.4%), the DFG 233,484 (92.2%), and the PDG 237,415.4 (93.8%), while the token- and AST-based approaches classified them all. Nevertheless, due to the possibility of combining several modules (cf. Section 4.3.1), JSTAP can still classify such samples.

4.2.2.2 `value` Features

In the second scenario, we consider the `value` approach. Contrary to the `ngrams` variant, the TPR and TNR are less constant across our five analyses, as shown in Figure 4.5, given that considering node value information increases the number of different features. In particular, the TPR ranges from 98.44% (AST) to 99.23% (CFG). Even though the CFG performs best to detect malicious JavaScript, it performs worse to accurately label benign samples, with a TNR of 95.87% compared to 99.67% for the token-based approach. The overall detection rates across the five analyses are also more sparse than with the `ngrams` approach: from 97.55% for the CFG to 99.44% for the lexical analysis, while AST, DFG, and PDG have similar detection rates between 98.9% and 99.1%.

This time and contrary to the `ngrams` variant, the lexical level of code abstraction leverages context information, since the `value` approach takes the value of each token into consideration (cf. Section 4.1.2.2). Thus, the `for` and `if` code snippets, that have an identical abstraction for the `ngrams` approach, have a different `value` representation, which contributes–for the reasons mentioned previously–to a better overall detection accuracy. In particular, each token has a value by construction, while only the `Identifier` and `Literal` nodes have one in the graph representations. For this purpose, we mapped the non-identifier and non-literal nodes to their nearest `Identifier/Literal` child, if any (on average, only 2.8 samples did not have any `Identifier` nor `Literal` nodes [157], representing 0.001% of our dataset). As a consequence, the same value is used by several nodes and may not always be informative, even though it is significant w.r.t. χ^2. Besides, the syntactic analyses do not fully benefit from the JavaScript grammar anymore, as each feature is analyzed independently (compared to an `ngrams` analysis, which combined n units). As mentioned in Section 4.2.2.1, the context information was mainly responsible for the high detection results; therefore, the lexical analysis now performs best. To overcome the lack of context, we tried to merge the current `value` approach with an n-gram analysis, by combining pairs of *(unit, value)* n times, but the TNR dropped to 80%. The features got indeed too specific to one file and could not be generalized over the whole dataset anymore. Last but not least, we assume that the CFG approach does not perform as well as the other ones, since benign and malicious developers may tend to use similar names for nodes with control flow edges as also suggested by the smaller number of features, compared to the AST or PDG, in Table 4.2.

4.2.2.3 Summary: Accuracy of JSTAP Modules

To sum up, our ten modules could all correctly classify our JavaScript collection with an accuracy over 97.55%; eight modules even have an accuracy over 99%. For the `ngrams` approach, the AST performs best because of the context information brought by the combination of syntactic units. Similarly, the `value` lexical module performs best thanks to the context information brought by the token values. Nevertheless, the CFG, DFG, and PDG are also very accurate ways of detecting malicious JavaScript samples and add all the more semantic information into the considered features.

4.2.2.4 Most Important Features for Classification

To accurately distinguish benign from malicious JavaScript inputs over 97.55% of the time, JSTAP leverages differences between benign and malicious samples at several abstract levels. Specifically, using the way in which given lexical and syntactic units are arranged in JavaScript files, along with their frequency, provides valuable insight to capture the salient properties of the code and identify recurrent patterns, specific to malicious vs. benign intent. For our ten modules, we extracted the five features most representative of malicious vs. benign intent, according to the corresponding random forest models [178]. For example, the most representative feature of the `ngrams` approach and for the AST, CFG, DFG, and PDG abstraction levels is the following: `[MemberExpression, MemberExpression, Identifier, Identifier]`, which is in line

with the tokens most representative feature, namely `[Punctuator, Identifier, Punctuator, Identifier]`, and represents an element of the form *a.b.c*. We assume that this construct is rather typical of benign samples, such as jQuery, which define several objects with multiple properties, while our malicious files rather store data inside simpler variables or arrays. For instance, the fourth most representative feature of the `value` tokens module is (`Punctuator, "+"`), which might point to the string splitting/string concatenation obfuscation technique. While it is massively used in malicious samples to evade, e.g., signatures-based detection, benign inputs might rather tend to avoid it due to the resulting performance downgrade. Similarly, the fifth most important feature of the `value` AST module is (`NewExpression, "Array"`), which may this time point to the obfuscation technique where strings are fetched from a global array.

All in all, malicious JavaScript samples try to hide their maliciousness by using different obfuscation techniques, which leave specific and recognizable traces in the source code. While benign documents may also be obfuscated to protect code privacy and intellectual property, they have more concerns about performance; thus, they use different techniques. For this reason, we also assume that malicious code is so different from benign inputs that the natural evolution of the code experienced over a few years should not change the detection results [S1]. Therefore, we consider that even if our malicious (Table 4.3) and benign (Table 4.4) datasets have not been collected at the same time, this does not introduce a bias in our experiments. Still, we discuss the extent to which attackers could craft malicious samples exactly reproducing a benign feature distribution in the following Chapter 5.

4.2.3 Analysis of Closely Related Detectors

Several systems already combined differences at a lexical or at an AST level with off-the-shelf supervised machine learning algorithms to distinguish malicious from benign JavaScript inputs. In this section, we focus on Cujo [173], Zozzle [50], and JaSt (Chapter 3), as they are–to the best of our knowledge–the most closely related work to our token- and AST-based approaches. We first present these detectors before comparing their implementation with the respective, conceptually similar, JStap module. We then highlight the overall better detection performance of JStap on our dataset as well as on samples likely to be trying to evade detection.

4.2.3.1 Presentation of Cujo, Zozzle, and JaSt

In 2010, Rieck et al. developed Cujo, which extracts n-gram features from JavaScript lexical units before using an SVM classifier for an accurate malware detection. As the system is not open source, we contacted the authors who pointed us to the tokenizer they initially used [172] and encouraged us to leverage the `HashingVectorizer` implementation from Scikit-learn [177] to map the extracted features to a corresponding vector space. In the original implementation, Cujo also leverages an enhanced version of ADSandbox [54], which executes the code associated with a web page within the JavaScript interpreter SpiderMonkey [148]. We contacted Dewald et al., who informed us that ADSandbox is neither maintained nor running anymore. Since we specifically

focus on *static* JavaScript detectors, we consider only the static part of Cujo. Also, we assume that our reimplementation is functionally equivalent to the original one, and, for reproducibility reasons, we make this system publicly available [T5].

In 2011, Curtsinger et al. implemented Zozzle, which combines the extraction of features from the AST, as well as their corresponding node value, with a Bayesian classification system to detect malicious JavaScript instances. We approached the authors and asked for their code or some inputs but did not get any response. Thus, we reimplemented the system with automatic feature selection, 1-level features, and multinomial naive Bayes, based on the information from their paper. Similarly to Cujo, Zozzle also has a dynamic part, to first hook into the JavaScript engine of a browser to get the deobfuscated version of the code. As previously, we reimplemented the static part of the tool, and we make it publicly available [T6].

Last but not least, we presented our system JaSt–which leverages n-gram features from the AST to detect malicious JavaScript instances–in the previous chapter. As our tool is open source [T1], we directly used it for the comparisons.

4.2.3.2 Benefits of JStap Modules

Conceptually, JStap's `ngrams` tokens module is identical to Cujo. In contrast, we rely on Esprima for the tokenization process, use 4-grams instead of 3-grams, do not consider all features, but select them with a χ^2 test, and use a different classifier (random forest). For Zozzle, JStap's `value` AST module is conceptually equivalent. Still, we consider all nodes from the AST (and not only expressions and variable declarations), a different confidence for the χ^2 test, and random forest instead of naive Bayes. As for JaSt, it is conceptually identical to JStap's `ngrams` AST module. Still, we do not simplify the syntactic units returned by the parser but perform a χ^2 test to reduce the size of our feature space.

4.2.3.3 Comparison and Added Value of JStap

Overall, and as presented in Figure 4.6, the three corresponding modules of JStap have a better detection performance than Cujo, Zozzle, and JaSt. Specifically, JStap has a higher TPR than Cujo (98.73% compared to 98.61%) and a higher TNR (99.4% vs. 97.9%), meaning that we classify 2,051 files more accurately than Cujo. We assume that our module performs better, given the differences we highlighted in the previous section. In particular, 4-grams performed better than 3-grams, and random forest performed better than SVM during our hyper-parameter selection process. Also, we hypothesize that Cujo performs differently than in its original paper (with a FPR of 2.0E-3% and 5.6% FNR) mainly due to our malicious dataset, comprising 131,448 samples from different sources, compared to 609 for Cujo originally. This way, our reimplementation recognizes more malicious JavaScript instances than initially but to the detriment of benign samples.

We observe a similar trend for Zozzle, which has a significantly lower TPR (94.27% vs. 98.44%) and TNR (97.35% vs. 99.54%) than the corresponding JStap module, for the reasons mentioned in the previous section. We also assume that Zozzle performs differently than in its original paper (with a FPR of 3.1E-4% and 9.2% FNR) due to our

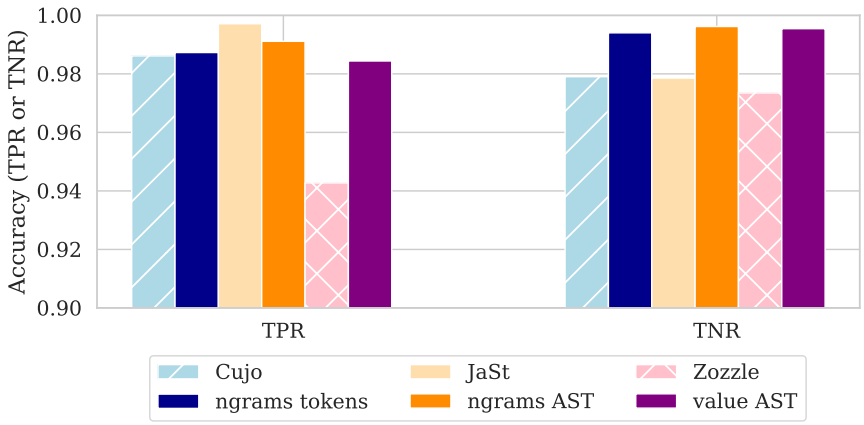


Figure 4.6: Comparison of our detection performance with related work

malicious dataset. Specifically, Curtsinger et al. considered only 919 malicious samples and clearly stated in 2011 that "relatively few identifier-renaming schemes [were] being employed by attackers", which is not the case anymore, since malicious samples are heavily obfuscated (as also observed during the manual analysis from Section 4.2.1.1). While it might be unfair to consider only the static parts of Cujo and Zozzle to compare these tools with the corresponding JStap modules–as their accuracy might also stem from their dynamic components–we aim at comparing several *static* analysis systems, working at different abstract levels (also, we did not have the original systems to check the added value, or not, of their dynamic components).

Finally, JaSt has a slightly higher TPR than JStap (99.71% vs. 99.11%) but in compensation a lower TNR (97.86% vs. 99.62%), meaning that JStap classifies on average 1,592.8 files more accurately. We believe that the higher accuracy of JStap is mainly due to us not simplifying the syntactic units returned by the parser this time (i.e., with JaSt, we grouped units with the same abstract meaning, e.g., we considered both a `ForStatement` and an `IfStatement` as a *Statement* node, therefore losing context information). Also, we assume that JaSt's results slightly differ from the previous chapter because of our dataset that is bigger and contains more diverse JavaScript samples than previously, which is in line with the assumptions we made for Cujo and Zozzle.

4.2.3.4 Samples with Conflicting Labels

As a final comparison step, we study the detection performance of JStap's modules on samples for which Cujo, Zozzle, and JaSt made different predictions. Given the different classification results, such samples may be trying to evade detection. Specifically, the three detectors made different predictions for 17,178.6 samples (6.78% of our dataset), 7,943.4 of which are malicious.

Table 4.5 presents the detection performance of all JStap modules, and of Cujo, Zozzle, and JaSt, on such samples. First, our `ngrams` tokens approach performs better than Cujo in this configuration too, with both a significantly higher TPR (87% vs.

Tool	Module	TPR	TNR	Accuracy
JStap	`ngrams` tokens	0.87	0.94	0.91
	`ngrams` AST	0.89	0.96	0.93
	`ngrams` CFG	0.91	0.93	0.92
	`ngrams` DFG	0.90	0.92	0.91
	`ngrams` PDG	0.90	0.93	0.92
	`value` tokens	0.89	0.96	0.93
	`value` AST	0.84	0.96	0.90
	`value` CFG	0.90	0.70	0.81
	`value` DFG	0.88	0.88	0.89
	`value` PDG	0.88	0.90	0.89
Cujo	-	0.78	0.64	0.71
Zozzle	-	0.14	0.66	0.42
JaSt	-	0.97	0.73	0.84

Table 4.5: Comparison of our detection performance with related work, on samples with conflicting labels

78%) and TNR (94% vs. 64%). Similarly, we outperform Zozzle by correctly classifying over twice as many samples with JStap `value` AST module (overall detection accuracy of 90% vs. 42%). Still, these results have to be taken with a grain of salt, as we tested the classifiers on samples likely to try to evade detection. In fact, in Section 4.2.3.3, Zozzle did not perform as well as Cujo and JaSt. In particular, it reported almost 7,000 false negatives (FNR of 5.7%) compared to 348 for JaSt and 1,675 for Cujo. Therefore, and out of the 7,943.4 malicious samples considered here, at least 5,300 are initial false negatives from Zozzle, meaning that its TPR could not be over 33%. Finally, and as previously, JaSt has a higher TPR than our `ngrams` AST approach (97% vs. 89%) but at the same time significantly fewer true negatives (73% vs. 96%), meaning that JStap has a higher overall detection accuracy, classifying 1,550 files more accurately than JaSt.

As for the remaining JStap modules, they are also impacted by these samples likely to be evasive. Specifically, they have a mean accuracy between 81% (`value` CFG) and 93% (`value` tokens), compared to over 97.55% (`value` CFG) and up to 99.44% (`value` tokens) in Section 4.2.2, on standard datasets. Overall, and even in this specific configuration, all JStap modules significantly outperform Cujo, Zozzle, and JaSt.

4.2.4 Run-Time Performance

Besides accurate predictions, we further evaluate the applicability of JStap in practice by focusing on its run-time performance. For this purpose, we measured its throughput on a server with four Intel(R) Xeon(R) Platinum 8160 CPUs (each with 48 logical cores) and a total of 1.5 TB RAM. Since JStap runs the analysis of each JavaScript file on a single core, the run-time reported is for a single CPU only. Table 4.6 presents the average, median, minimum, and maximum duration to generate each of our considered code representation. The tokenizing and parsing with Esprima are relatively fast, with an average time of 17 and 35 ms per file. The most time-consuming operation is the

Code representation	Mean (ms)	Median (ms)	Min (ms)	Max (s)
Tokens	16.894	9.000	0.000	0.175
Parsing	34.921	19.000	1.000	0.311
AST from parsing	97.711	11.487	0.038	4.103
CFG from AST	39.085	4.635	0.004	1.114
PDG from CFG	369.490	8.710	0.125	27.270

Table 4.6: Run-time performance to generate JSTAP's code representations

Module	Feature extraction				Random forest	
	Mean (ms)	Median (ms)	Min (ms)	Max (s)	Train (ms)	Classify (ms)
`ngrams` tokens	2.344	1.420	0.650	0.203	0.162	0.715
`ngrams` AST	9.683	2.592	0.635	0.722	0.190	1.427
`ngrams` CFG	18.288	3.781	0.762	0.778	0.252	1.667
`ngrams` DFG	19.412	3.736	0.723	1.111	0.228	2.685
`ngrams` PDG	34.745	5.544	0.799	1.243	0.241	2.763
`value` tokens	13.251	3.743	0.947	1.397	0.187	1.127
`value` AST	112.036	11.131	1.085	86.619	0.227	1.370
`value` CFG	129.770	12.138	0.875	207.255	0.187	1.174
`value` DFG	101.830	9.707	0.990	107.432	0.195	1.279
`value` PDG	216.895	21.440	1.003	247.253	0.173	1.311

Table 4.7: Run-time performance of JSTAP per module

PDG generation, which highly depends on the AST size, since we have to traverse it, pushing and popping the variables encountered all the way down to the leaves. For performance reasons, we generated the PDGs of all files from our dataset once, and we stored the PDGs to not have to produce them for each module again. Therefore, we did not take into consideration the PDGs (and tokens, for comparison purpose) generation time in Table 4.7.

This table presents the duration to generate the features considered by each module, for *one* file. The last two columns stand for the run-time to leverage these features to train our random forest classifier (averaged for one file) and to classify one unknown input. Overall, we note that more complicated code representations (e.g., PDG and CFG, compared to tokens and AST) lead to a higher overhead, since we follow more edges in the graphs and consider more features. The `value` approach also is slower than the `ngrams`, as we fetch a value for each unit, thereby traversing sub-ASTs down to the leaves each time.

Specifically, classifying a JavaScript sample with the `ngrams` tokens module takes on average 19 ms for the feature generation (including tokens production) and 0.71 ms for the classification. For the `value` AST approach, it takes 112 ms to produce features, with an AST previously generated, and 1.4 ms for the classification. Based on the number of features each module considers (cf. Table 4.2), and an average size of 23 KB per file, we consider the overhead to be reasonable. Also, JSTAP is fully parallelized to leverage all available CPU cores for a faster analysis for a deployment in the wild.

4.3 Combining JSTAP Modules

JSTAP is a modular malicious JavaScript static detection system for which the user can choose the analysis type (`ngrams` or `value`) as well as its abstraction level (tokens, AST, CFG, DFG, or PDG). We highlighted previously the high accuracy of each JSTAP module independently (except for `value` CFG, between 98.9% and 99.44% of the predictions are correct). In this section, we combine several modules to simultaneously leverage different ways to abstract JavaScript code, leading to a higher detection accuracy. To this end, we finally discuss combining modules to pre-filter samples with conflicting labels and send only those few to more costly follow-up analyses.

4.3.1 Module Combination

First, we justify our choice to combine the `ngrams` AST, `value` tokens, and `value` PDG modules. Then, we present our combination pipeline, where we take the prediction with the most votes to classify a given input. Finally, we report on the proportion of samples for which all three modules made the same predictions and discuss our accuracy both on these samples and on the remaining ones.

4.3.1.1 Module Selection

For the combination process, we chose the `value` tokens and `ngrams` AST approaches. Both of them are indeed very accurate (99.44% and 99.38% correct predictions, cf. Section 4.2.2). In addition, they use different features that do not overlap. In fact, the former leverages the lexical structure of a JavaScript file and combines each extracted token with its corresponding value, while the latter rests upon an AST traversal and an n-gram combination of the traversed nodes for an accurate malicious JavaScript detection. To perform majority voting in terms of module predictions, we need an odd detector number. The `value` PDG approach then complements the `value` token and `ngrams` AST systems, as it also uses new features, which do not overlap with the previous ones (i.e., combination of control and data flows with node value information).

4.3.1.2 Majority Voting

Subsequently, we classify our datasets presented in Section 4.2.1 by combining the predictions of the three selected modules (`ngrams` AST, `value` tokens, and `value` PDG). In particular, for a given JavaScript input, we chose the prediction with the most votes. Such a combination presents both a high TPR of 99.2% and a TNR of 99.7%, representing an accuracy of 99.46%. Still, when we considered each module separately in Section 4.2.2, we had an approaching accuracy for the `value` tokens module (best one) with a detection rate of 99.44%. Thus, combining modules leads to a detection of 36 additional samples (0.015% of our dataset), which we do not see as a major improvement.

Nevertheless, we also leveraged the combination of these three modules to specifically classify the 17,178.6 samples for which CUJO, ZOZZLE, and JAST made different predictions, i.e., samples potentially trying to evade detection (cf. Section 4.2.3.4).

This time, we retain an accuracy of 93.47%, which is, again, better than each module separately. In particular, with this combination of modules, we detect on average 47.9 extra samples compared to the `value` tokens approach, which is 1.3 times more samples than previously with our standard dataset that contains almost 15 times more samples. Therefore, combining modules brings a real added value when classifying samples likely to be evasive. Similarly, this combination also enables us to correctly classify 121.9 extra samples compared to the `ngrams` AST approach. In particular, the `value` tokens module correctly classifies 74 extra samples compared to the `ngrams` AST variant (0.43% of our evasive dataset), while only classifying 0.07% more of our standard dataset. Thus, the difference in terms of detection accuracy between these two modules tends to increase on evasive samples. For this reason, combined JSTAP modules perform better than each module separately, in particular on evasive samples. In the case of evasive samples, some modules may struggle to classify them correctly, while combining modules limits the proportion of samples evading our system.

4.3.1.3 Unanimous Predictions

Last but not least, we focus on the JavaScript samples for which all three of our combined modules made the same predictions, and on the contrary, those for which they had different classification results. On average, `ngrams` AST, `value` tokens, and `value` PDG labeled 234,875.8 JavaScript inputs the same way (92.76% of our dataset). On these samples specifically, we have both an extremely high TPR of 99.55% and TNR of 99.9% (standing for an overall detection accuracy of 99.73%). Since modules already performed better with majority voting compared to each module separately, we naturally expected this higher accuracy when all three modules make the same predictions.

Finally, we classified the remaining 18,340.2 samples with conflicting labels (over 80% of which are benign), which can also be seen as samples trying to evade detection (similarly to Section 4.2.3.4). Still, we retain a high TNR of 98.16% on these samples, with the majority voting system. In turn, we have a TPR of 86.9%, leading to an accuracy of 96% on samples for which our combined detectors predict conflicting labels, which we consider to still be relatively high. Also, the overall detection accuracy of JSTAP should not be evaluated only on such samples but on our whole dataset (also containing these samples), where we retain an accuracy of almost 99.5% (cf. Section 4.3.1.2).

4.3.2 Improving the Detection with Pre-Filtering Layers

To detect malicious JavaScript inputs, we specifically chose to perform a *static* analysis, which is by construction fast, while still making accurate predictions. Dynamic detectors may perform better, in particular, if they visit all possible execution paths [105, 108]. Still, they are more costly, as they require specific instrumentations, introduce overhead inherently depending on the code execution, and the necessary amount of time to observe a malicious behavior is not defined [190]. Also, such dynamic analyses can be defeated if samples notice that they are running in a sandboxed environment [11, 23].

To both maximize the detection accuracy and minimize the run-time performance, we rather envision that JSTAP could be used to *pre-filter* JavaScript samples, sending, e.g., only those with conflicting labels to much slower dynamic components. In the context

of Section 4.3.1.3, the 18,340.2 samples (on average) for which `ngrams` AST, `value` tokens, and `value` PDG made different predictions could be sent to such components. There is naturally a trade-off (which may be user specific) to find between accuracy and run-time performance. In fact, we could also consider a second pre-filtering layer to further limit the number of inputs to be executed in a sandboxed environment. Similarly to the combination of Cujo, JaSt, and Zozzle, we combined `ngrams` tokens, `ngrams` AST, and `value` AST to classify the 18,340.2 samples we considered previously. These three detectors predicted the same labels for 16,469.4 inputs, with an accuracy of 99%, meaning that only the resulting 1,870.8 samples could be sent to dynamic components. This way, out of our 253,216 sample set, JStap correctly classified 234,875.8 instances (92.76%) with an accuracy of 99.73% in a first pre-filtering step (cf. Section 4.3.1.3). Then, it correctly labeled 16,469.4 additional samples (6.5% of the initial dataset) with an accuracy of 99% in a second pre-filtering step. Thus, in this configuration, our detection accuracy lies significantly over 99% for 99.26% of our original dataset, and only the remaining 0.74% of this dataset would be outsourced to more costly components.

4.4 Summary

In this chapter, we presented JStap, our modular static malicious JavaScript detector, which goes beyond leveraging purely lexical and syntactic information. In fact, and contrary to JaSt, JStap also considers semantic information in the form of control and data flows (*RQ2*). Specifically, we propose ten modules with differing levels of context and semantic information, which we combine with a random forest classifier to detect malicious JavaScript instances. In practice, our modules are all, and independently, very accurate; the best one having an accuracy of 99.44%, with a low false-positive rate of 0.33% and 0.8% false negatives. For JStap to make more accurate predictions, we then combined the output of three modules that leverage complementary code abstractions (i.e., tokens, AST, and PDG) so as to use different aspects of the samples for classification. This way, we envision that this combination of modules could be used as a pre-filtering step before sending samples with conflicting labels to more costly follow-up analyses. In this scenario, we could classify almost 93% of our dataset with a detection accuracy of 99.73%. A second combination of modules then enabled us to classify a remaining 6.5% of our dataset with an accuracy still over 99%. Thus, with these two JStap pre-filtering layers, less than 1% of our initial dataset would require additional scrutiny.

While JaSt and JStap are both very accurate, they leverage learning components for their predictions. By construction, such machine learning-based detectors will fail to detect some attacks, e.g., if a malicious sample has been manipulated so that its extracted features map a benign distribution. In the next chapter, we go one step further and propose a generic attack against static classifiers, where we rewrite ASTs of malicious JavaScript samples to exactly reproduce existing benign ones.

5

HIDENOSEEK: Camouflaging Malicious JavaScript in Benign ASTs

In the previous chapter, we presented our static and modular malicious JavaScript detector JStap, which goes beyond traditional lexical- and AST-based pipelines by also considering control and data flow information. As JaSt, JStap relies on machine learning algorithms to classify JavaScript inputs. Still, the field of attacks against machine learning-based approaches is vast. In particular, it has been shown that attackers with *specific* and *internal* knowledge of a target system may be able to produce input samples, which are misclassified. In practice, the assumption of *strong attackers* is not realistic, as it implies access to insider information.

In this chapter, we consider *RQ3: Can we present a generic attack against static malicious JavaScript detectors? More specifically, to what extent and how could attackers rewrite the ASTs of malicious JavaScript samples to reproduce existing benign ASTs while keeping the original malicious semantics? How effective would this camouflage be against static detectors?* To investigate these research questions, we propose HideNoSeek, our camouflage attack, which, by design, evades the entire class of detectors based on syntactic features, without needing any information about the target systems. To rewrite malicious ASTs into existing benign ones, HideNoSeek first searches for isomorphic subgraphs between the considered malicious ASTs and benign ones. If it could find all malicious syntactic structures in a benign tree, it replaces the benign sub-ASTs with their malicious equivalents before adjusting the benign data dependencies to retain the malicious semantics. In practice, we leveraged 22 malicious seeds to generate 91,020 malicious scripts, which perfectly reproduce ASTs of Alexa top 10k websites. Finally, we evaluate these malicious samples against real-world static detectors and discuss potential mitigations against our attack.

5.1 Motivation

Due to their speed and accuracy, static malware detectors are particularly relevant to quickly discard benign samples, leaving only those few, which are likely malicious, for costly manual analysis or dynamic components. Therefore, static systems have to be accurate to neither waste expensive resources nor let malicious files through. Several approaches have been proposed to detect malicious JavaScript statically, such as Cujo [173], Zozzle [50], JaSt (Chapter 3), and JStap (Chapter 4). All of these detectors combine static features with machine learning algorithms. Due to their usage of static features, they represent a subset of the detectors HideNoSeek targets. In this section, we first present existing attacks against learning-based malware detectors and highlight their limitations. Next, we motivate our deobfuscation step to increase the success of our attack.

5.1.1 Limitations of Existing Attacks

As mentioned previously, the field of attacks against machine learning-based systems is vast [13, 14]. In fact, machine learning rests upon the assumption that training and test data follow the same distribution [9]. By design, machine learning pipelines are then vulnerable to crafted examples that violate the previous assumption. In practice, several attacks have been proposed to evade classifiers. Such attacks transform a

given input sample so that it keeps its intrinsic properties, but the targeted classifier's predictions between the original and the modified input differ, e.g., adversarial attacks on PDF [129, 183, 190] or Android malware detectors [53, 72, 166], spam filtering [127], or mutations of malicious samples [51, 228]. However, for these attacks to be effective, malicious actors need information about the classifier they were trying to evade, e.g., some knowledge about the training dataset or the target model internals, or at least the classification scores assigned to input samples. Another class of attacks focuses on the transferability in machine learning. In fact, adversarial examples affecting one model often affect another, even if they have different architectures or training sets, provided they were trained to perform the same task. Therefore, attackers can also build and train a surrogate classifier, craft adversarial examples to evade their system before transferring the samples to the victim classifier [161, 162, 189]. Still, to train their own classifier, attackers need a specific target system as well as access to it.

By design, all previous attacks are inherently limited because attackers need internal and specific information about the detector they are trying to evade. On the contrary, HideNoSeek works independently of any machine learning system and does not need any knowledge of model internals, training dataset, or a specific classifier to test. In essence, our system crafts malicious samples, which have exactly the same AST as existing benign JavaScript files. Therefore, and by construction, our approach directly foils AST-based detectors. Furthermore, we showcase that our attack is also effective against detectors leveraging tokens, control, and/or data flow information, as well as anti-virus systems using structural analysis, e.g., signatures or content-matching. Due to the exact mapping onto a benign AST, our attack is also naturally more effective than existing malware, which is, e.g., inserted in bigger benign files, to evade detection by statistically increasing the proportion of benign features.

5.1.2 Malicious JavaScript Deobfuscation

HideNoSeek leverages the fact that malicious obfuscation leaves specific traces in the syntax of malicious files, which enable to differentiate malicious instances from benign (even obfuscated) inputs (as highlighted in the two previous chapters). Instead of trying to hide the maliciousness of a JavaScript file behind traditional obfuscation layers, which contribute to their detection, HideNoSeek changes the constructs of malicious samples to reproduce an existing benign syntax (this camouflaging process can also be seen as a new form of obfuscation). As a consequence, HideNoSeek automatically foils any classifier leveraging the syntactic structure for a malicious JavaScript detection, i.e., most of the detectors previously outlined. In essence, HideNoSeek specifically crafts samples that are likely to be labeled as benign by static pre-filtering systems, meaning that they would not be analyzed by dynamic components.

The core of our attack consists in rewriting malicious ASTs into existing benign ones. To this end, we first look for isomorphic subgraphs between malicious and benign ASTs. Since malicious obfuscation is responsible for differences at the AST level, we first deobfuscated our malicious files, to get the original payload (i.e., the original syntax), which resembles more benign ASTs than the obfuscated versions. We combined JSDetox [200] and box-js [28] with a manual analysis for the deobfuscation process. In

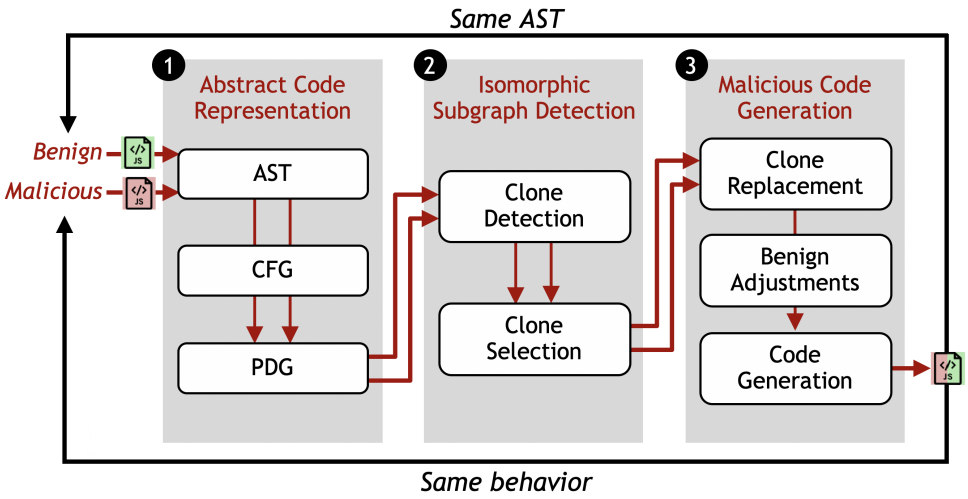


Figure 5.1: Architecture of HIDENOSEEK

the following, we use the term *seed* to refer to an original malicious payload, i.e., the state we assume a malicious entity had before obfuscation or packing.

5.2 Rewriting a Malicious AST into an Existing Benign One

HIDENOSEEK aims at automatically rewriting the AST of a malicious JavaScript input into an existing benign one while retaining the malicious semantics after execution. In its core, we implemented HIDENOSEEK in Python. We first provide a high-level overview of our system before discussing its three main components, namely an abstract code representation part, a clone detector, and a malicious code generator, into more details. Across this section, we illustrate our approach with the following simplified example: we explain how we crafted the resulting sample from Listing 5.3 by rewriting the AST of the malicious seed from Listing 5.1 to reproduce the AST of the benign sample from Listing 5.2.

5.2.1 HIDENOSEEK Conceptual Overview

As illustrated by Figure 5.1, HIDENOSEEK takes a malicious seed m and a benign document b as input, and outputs a sample s with the same AST as b while retaining the malicious semantics of m.

First, we perform a static analysis of the benign and malicious JavaScript samples considered and augment their respective ASTs with control and data flow information (Figure 5.1 stage 1). The resulting two graphs enable to reason about variable dependencies, the order in which statements are executed, as well as the conditions that have to be met for a specific execution path to be taken (Section 5.2.2). Subsequently, HIDENOSEEK uses these two graphs to look for identical sub-ASTs between the malicious seed and the considered benign input (Figure 5.1 stage 2). For this purpose, HIDENOSEEK looks for pairs of matching benign and malicious statement nodes (i.e., same abstract syntactic structure) and slices backward along their control

and data flow edges as long as it reports further matching statements. We store these common syntactic structures together in a list (slice) and refer to them as *clones*. Since a malicious sub-AST may be found several times in a given benign input, we define criteria to, e.g., maximize the clone size or minimize the distance between the nodes inside a clone, thereby reducing the adjustment surface (Section 5.2.3). In fact, HIDENOSEEK replaces the benign clones by the malicious ones and follows the original benign data flow edges to automatically adjust the initial benign nodes–which were impacted by the replacement process–for them to still respect the original benign AST structure while keeping the malicious semantics after execution (Figure 5.1 stage 3). Finally, we transform our crafted AST back to code (Section 5.2.4).

This way, HIDENOSEEK generated a sample, which reproduces the AST of an existing benign input while retaining the semantics of a malicious file.

5.2.2 PDG Generation

To detect identical benign and malicious sub-ASTs, with respect to control and data flows, HIDENOSEEK is based on an abstract, labeled, and directed code representation. The AST provides both a hierarchical decomposition of the source code into syntactic elements and code abstraction, ignoring, e.g., the variable names and values to consider them as `Identifier` and `Literal` nodes. In addition, we add control and data flow edges to the AST to also take into account execution path conditions and the semantic order of the code. We use the term PDG to refer to the resulting graph.

5.2.2.1 AST Generation

Similarly to Section 4.1.1.2, HIDENOSEEK leverages the parser Esprima to generate the AST of a valid JavaScript sample. Still, to detect syntactic clones (in particular if they are not contiguous), we also need control and data flow information.

5.2.2.2 CFG: AST + Control Flow

For this purpose, we extend the AST with control flow edges, as described in Section 4.1.1.3. As previously, we use the term CFG to refer to the resulting graph.

5.2.2.3 PDG: AST + Control Flow + Data Flow

Finally, we also add data flow edges to the CFG, as described in Section 4.1.1.4. We refer to the resulting graph as a PDG.

5.2.3 Slicing-Based Clone Detection

Given the space $\mathcal{B}$ of benign JavaScript samples and the space $\mathcal{M}$ of malicious ones (according to some oracle), we aim at building a sample space $\mathcal{S}$ so that:

$$\mathcal{S} = \{x | x \in \mathcal{M}, \exists x' \in \mathcal{B} | ast(x) = ast(x')\}$$

with $ast(x)$ the AST of the sample x .

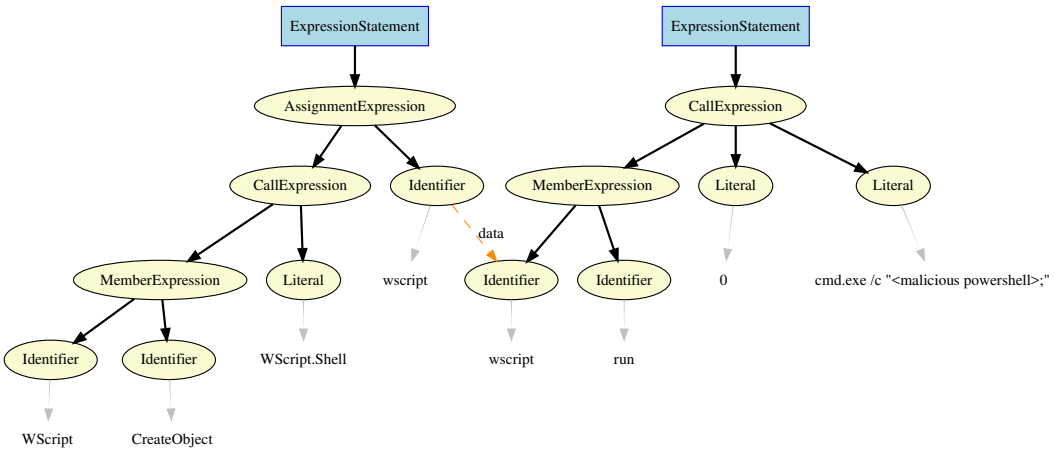


Figure 5.2: AST of Listing 5.1 (malicious) extended with control & data flows

To this end, we first look for sub-ASTs from a malicious file that can also be found in a benign one. We refer to such common structures as *clones*. To detect clones, we consider the algorithm of Komondoor et al. [110], which combines PDGs with a variation of program slicing [83, 217]. Several algorithms have been proposed to detect clones, e.g., based on tokens [112], the AST [15], and the PDG [114]. We chose the approach of Komondoor et al., as the addition of the slicing part enables us to find non-contiguous syntactic clones and clones in which matching statements have been reordered. First, we create equivalence classes (Section 5.2.3.1), which regroup common benign and malicious AST statement nodes based on their abstract syntactic meaning. Then, for each benign and malicious pair from the same class and with the same AST edges[1] (i.e., slicing criterion), we add the pair to the current clones list and slice backward along the control and data flow edges of the corresponding nodes. We add their predecessors to the current clones list if and only if they match (i.e., same sub-AST), and we iterate as long as HideNoSeek finds matching statement nodes (Section 5.2.3.2). Finally, as a malicious sub-AST may be found several times in the same benign document, we define criteria to select the strongest clones (Section 5.2.3.3).

5.2.3.1 Equivalence Classes

Finding clones between a benign and a malicious file consists in finding isomorphic subgraphs between their abstract syntactic representations. Computing all pairs of benign and malicious statement nodes and comparing their syntax, to keep only the matching nodes, would not be efficient. Thus, HideNoSeek first partitions the benign PDG statement nodes (displayed with blue squares in the graphical representations) into equivalence classes based on their syntactic structure. For example, the PDG of Figure 5.3 has only one class, namely `ExpressionStatement`, containing five elements. Then, we complete the equivalence classes with statement node information

[1] For example, two statement nodes with the same name, e.g., `IfStatement`, may have a different sub-AST, meaning that they are no clone; thus, not relevant for us

```
Data: benign statement node benign, malicious statement node malicious, clones found so far
      clones_list
Result: clones_list updated with the clones found in benign and malicious
initialization;
if benign and malicious belong to the same equivalence class and have the same sub-AST then
    if they have already been handled together then
        search the corresponding clones_list entry;
        append to this entry the clones found so far;
    else
        create a new clones_list entry;
        add benign and malicious to this entry;
        benign_parents ← backward_slice(benign);
        malicious_parents ← backward_slice(malicious);
        iterate over benign_parents and malicious_parents;
        call find_clone on the resulting combinations;
    end
else
    benign_parents ← backward_slice(benign);
    iterate over benign_parents;
    call find_clone(benign_parents, malicious);
end
return clones_list
```

Algorithm 5.1: *find_clone*: finds isomorphic subgraphs from two statement nodes

from the considered malicious file. Specifically, the PDG of Figure 5.2 adds two malicious elements in the `ExpressionStatement` class. At this stage, a benign and a malicious node from the same class do not necessarily match, as they could have a different sub-AST. For instance, the different `ExpressionStatement` nodes do not all have the same syntactic structure. We perform this verification in the next step.

5.2.3.2 Clone Detection

Next, we iterate through the equivalence classes list: for each equivalence class, we call the *find_clone* function, described in Algorithm 5.1, on every benign and malicious pair *(b, m)*. To find two isomorphic subgraphs, the former containing b and the latter m, HideNoSeek verifies that they have the same sub-AST by traversing and comparing their respective nodes. If they have an identical sub-AST, HideNoSeek adds the corresponding nodes to our current clones list. Then, HideNoSeek slices backward in lock step along the control and data flow edges, starting from b and m, and adds their respective predecessors to the current clones list if and only if they match (i.e., same class and same sub-AST). We iterate the process as long as we find predecessors that HideNoSeek have not handled yet. Because of backward slicing along the control and data flows, this algorithm can find non-contiguous clones (i.e., clones whose components do not occur directly one after the other in the source code) as well as clones in which matching statements have been reordered.

In addition, whenever we find a pair of matching statement nodes that HideNoSeek has already handled, the process stops for the current pair, the system retrieves the clones, which it found previously on the pair, and adds these nodes to the current

```
1 wscript = WScript.CreateObject('WScript.Shell');
2 wscript.run("cmd.exe /c \"<malicious powershell>;\"", "0");
```

Listing 5.1: Malicious JavaScript code example

```
1 obj = document.createElement("object");
2 obj.setAttribute("id", this.internal.flash.id);
3 obj.setAttribute("type", "application/x-shockwave-flash");
4 obj.setAttribute("tabindex", "-1");
5 createParam(obj, "flashvars", flashVars);
```

Listing 5.2: Benign JavaScript code example (extract of the plugin jPlayer 2.9.2)

slice. Besides performance improvement, it also ensures that we do not report any subsumed clone at this stage. Furthermore, when we test a pair of non-matching statement nodes *(b, m)*, the system still recursively slices backward from b and tests its predecessors against m which enables to jump over benign data dependencies to find more non-contiguous clones. Because of this step, we can find two isomorphic subgraphs, which are no ASTs, therefore expanding the possible set of clones.

For example, HideNoSeek detects that the ASTs of Listing 5.1 and Listing 5.2 match, respectively for the lines 2 and 3 (format: *a.b(str1, str2)*).[2] By slicing backward along the data dependencies, our system respectively tests the lines 1 and 2, which do not match. Applying the previous rule, it respectively tests the lines 1 and 1, which match (format: $a = b.c(str)$). This way, HideNoSeek found the complete AST of Listing 5.1 (malicious) in Listing 5.2 (benign).

When the clone detection process finishes, it has identified two isomorphic subgraphs: one benign and one malicious, which may contain several nodes. HideNoSeek may report further pairs of isomorphic subgraphs (independent of the previous one) while iterating through the equivalence classes, hence the need for some metrics to determine the strongest pair of clones.

5.2.3.3 Strongest Clone Selection

A portion of the same malicious AST can be found several times in the benign one (the opposite may also be true). Since HideNoSeek needs to replace only one benign sub-AST by a syntactically equivalent malicious one, it can select only one clone, in this case. The first criterion consists in choosing the largest clone (based on the number of matching statement nodes it contains), and not a subsumed version of it. This way, we maximize the clone coverage, knowing that HideNoSeek can report subsumed clones only when it jumps over a non-matching benign node to consider its data flow predecessors. For our attack to be effective, though, the *complete* malicious sample has naturally to be replaced in syntactically equivalent parts of the considered benign file.

[2] HideNoSeek works directly at the AST level. Still, for clarity reasons, we rather chose to illustrate our approach at the code level. The corresponding ASTs are in Figure 5.2 (malicious), Figure 5.3 (benign), and ultimately Figure 5.4 (crafted)

The second criterion consists in maximizing the proportion of identical tokens between the benign and the crafted samples. In fact, reproducing the AST automatically copies most of the tokens, but we may observe some differences for the syntactic unit `Literal`, which can be translated into several tokens, e.g., `Int`, `Numeric`, or `Null`, depending on the context. If some tokens do not match, HIDENOSEEK reports them and suggests how to modify them so that they match the original tokens, e.g., the `Bool` *false* is equivalent to the `String` *"0"*, and to the `Int` *0*. The third and last criterion consists in minimizing the distance between the nodes inside non-contiguous clones, to minimize the adjustment surface (cf. Section 5.2.4.2). In our example, HIDENOSEEK ultimately reports one clone, composed of two statement nodes, namely lines 1 and 2 of Listing 5.1 (malicious) have the same syntactic structure as lines 1 and 3 of Listing 5.2 (benign).

Naturally, HIDENOSEEK does not necessarily report clones for all *(b, m)* pairs tested, as they may have different syntactic structures. To maximize the number of syntactic clones detected, we semi-automatically generated (up to three) different syntactic versions of a given malicious seed (cf. Section 5.3.1.1). For example, the `VariableDeclaration` *var a = 10* (in top-level code), the `AssignmentExpression` *a = 10*, and the `ExpressionStatement` *window.a = 10* are semantically equivalent, but syntactically different.

5.2.4 Malicious Code with a Benign AST

Once HIDENOSEEK has found identical benign and malicious sub-ASTs, it replaces the benign sub-ASTs with the corresponding malicious ones (Section 5.2.4.1). This process then yields some adjustments for the modified AST to correspond to valid code, which is still able to run. Therefore, HIDENOSEEK follows the data flows originating from the initial benign nodes (which have been replaced by their malicious equivalent) to collect the nodes that have been impacted by our replacement process. This way, we can modify them, with generic transformations and with respect to the AST (Section 5.2.4.2), so that we ultimately generate valid code with the original malicious semantics (Section 5.2.4.3).

5.2.4.1 Clone Replacement

As mentioned in Section 4.1.1.2, an AST is composed of inner and leaf nodes, the latter, which represents the operands. Saying that a benign AST is identical to a malicious one means that they both have the same nodes, with the same AST edges. Still, the benign code is different from the malicious one, as the attributes of the benign vs. malicious AST's leaves are different, e.g., variable names are not directly contained in the AST, but are attributes of the leaves. Therefore, HIDENOSEEK replaces the attributes of the benign leaves with the attributes of the malicious ones. This way, we did not change the benign AST, but the portion we replaced will lead to the code of the malicious seed and not to the benign code anymore. For example, lines 1 and 3 of Listing 5.3 (crafted) illustrate the replacement of Listing 5.1 (malicious) in the corresponding part of Listing 5.2 (benign). Nevertheless, this replacement process has modified the benign environment, which might interfere with the benign functionalities and could result in the crafted sample not running anymore.

```
1 wscript = WScript.CreateObject('WScript.Shell');
2 wscript.toString('id', this.internal.flash.id);
3 wscript.run('cmd.exe /c "<malicious powershell>;"', "0");
4 wscript.hasOwnProperty('tabindex', '-1');
5 parseFloat(wscript, 'flashvars', flashVars);
```

Listing 5.3: Resulting crafted JavaScript code with the benign AST of Listing 5.2 and malicious semantics of Listing 5.1

5.2.4.2 Benign Adjustments

As a countermeasure, HideNoSeek searches for fragments that may have been impacted by the replacement process and automatically adjusts them to the environment so that the crafted AST will lead to code that still runs. To this end, our system recursively explores the data flows originating from benign clone nodes under the conditions that (a) they do not belong to a clone node, and (b) they have not been handled yet, e.g., lines 2, 4, and 5 of Listing 5.2 (benign). HideNoSeek first renames the benign variables, impacted by the replacement, with the name of the malicious variables, which are now part of the code, as indicated in the corresponding lines of Listing 5.3 (crafted). At the AST level, Figure 5.3 (benign) and Figure 5.4 (crafted) have the same AST, but we renamed the variable *obj* in *wscript*.

HideNoSeek then analyses the end of each data flow edge and recursively stores the nodes it contains in a list, all the way down to the leaves. After that, our system searches in its database for a sublist of the considered list of nodes. In fact, we aim at determining the generic modifications HideNoSeek could perform on the benign nodes for the code to still run while keeping its original AST structure. If HideNoSeek does not find a match in the database, it reports the missing pattern so that we can search for a new adjustment and add it to the database. For example, if [`CallExpression`, `Identifier`] is a sublist of the considered list of nodes, it means that the benign code would look like *func(my_obj, [params])*, where *func* and *params* respectively refer to a benign function with its parameters, and *my_obj* stands for the object the data flow points to, i.e., the object that HideNoSeek modified. Because of our replacement process, the function may not run anymore. To avoid this phenomenon, HideNoSeek replaces the original function name with a function, which can be executed for every possible parameter type and number, without throwing an error or causing side effects. Such functions include `decodeURI()`, `isFinite()`, and `parseFloat()`. Our current list contains nine different function names so that we can randomly select one each time that we need such an adjustment. Line 5 of Listing 5.3 illustrates this specific adjustment process. Other adjustments may include a property or a method called on an object we modified, e.g., lines 2 and 4 of Listing 5.3. As previously, HideNoSeek has a list of nine properties that can also be used as methods, such as `hasOwnProperty` and `toString`, which can be combined and are valid in all contexts. This way, we can automatically adjust the crafted AST with generic modifications to limit, e.g., undefined variables or run-time errors, in the resulting code.

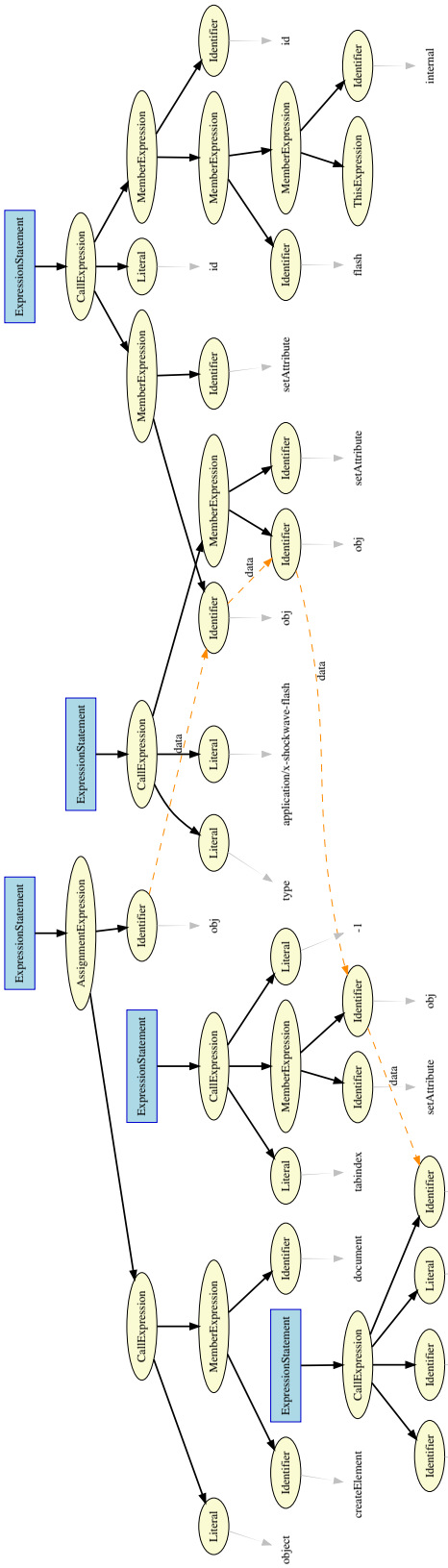


Figure 5.3: AST of Listing 5.2 (benign) extended with control & data flows

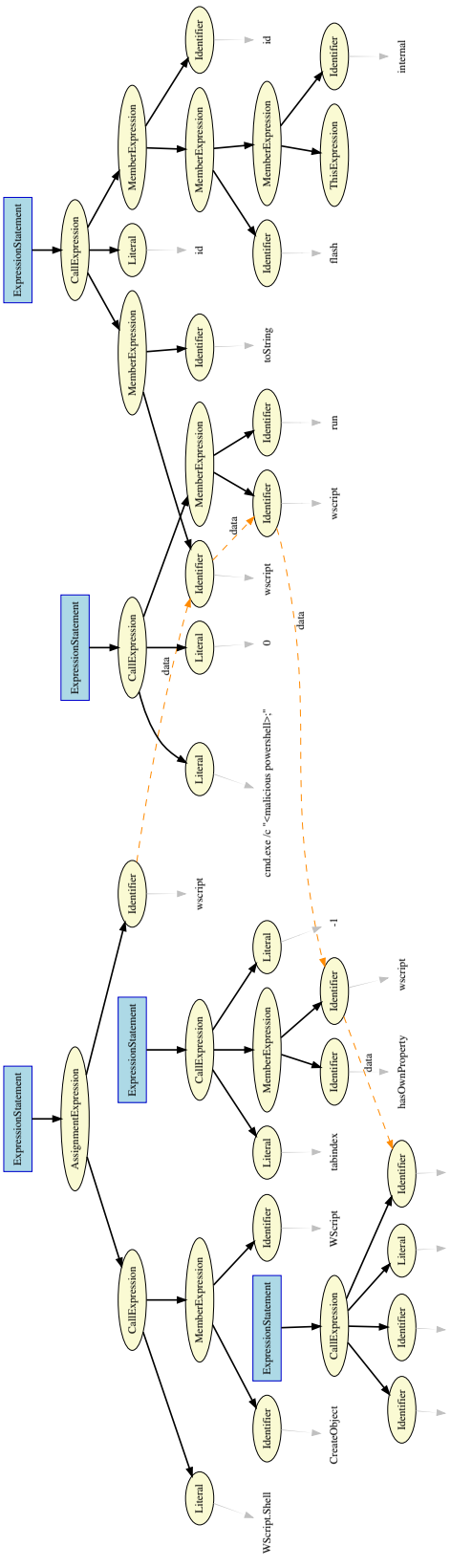


Figure 5.4: AST of Listing 5.3 (crafted) extended with control & data flows (i.e., rewrote the benign AST from Figure 5.3 with the malicious semantics of Listing 5.1

Source	Collection	#Samples	#Clusters	#Deobf samples
BSI	2017-2018	85,059	10	10
Hynek	2015-2017	30,247	15	15
DNC	2014-2018	4,444	9	11
GoS	2017	2,595	27	19
Sum	2014-2018	122,345	61	55
VirusTotal	2017-2018	13,884	8	8

Table 5.1: Malicious JavaScript dataset

5.2.4.3 Malicious Code Generation

Finally, to transform the crafted AST back to JavaScript code, we leverage the ECMAScript code generator Escodegen [199]. Specifically, we generated Listing 5.3 from its AST in Figure 5.4. By construction, the resulting code has the same AST as the original benign input from Listing 5.2 (i.e., Figure 5.3 and Figure 5.4 have the same AST) while keeping the malicious semantics of Listing 5.1.

5.3 HIDENOSEEK Samples

To highlight the feasibility of our attack, we first focus on the samples that HIDENOSEEK can generate (we test the crafted samples with different detectors in Section 5.4). For this purpose, we consider 23 unique malicious seeds (deobfuscated) and 8,546 different benign scripts. Our system then leverages these inputs to produce malicious samples that have the same AST as our benign scripts. We first evaluate the number of benign ASTs our system was able to reproduce per seed before considering the impact our attack would have. Subsequently, we verify the validity and maliciousness of the samples we crafted. Finally, we analyze HIDENOSEEK run-time performance.

5.3.1 Dataset Collection and Setup

To generate HIDENOSEEK samples, we leverage a malicious and a benign dataset, which we both describe in this section.

5.3.1.1 Malicious Dataset

Our malicious dataset, presented in Table 5.1, is a collection of 122,345 samples collected between 2014 and 2018 (73% of which have been collected after 2017). It includes, in particular, exploit kits provided by Kafeine DNC (DNC) [100] and GeeksOnSecurity (GoS) [68] as well as the malware collection of Hynek Petrak (Hynek) [86], and samples from the German Federal Office for Information Security (BSI) [24]. We consider that all these files are malicious. In fact, the deobfuscation and the manual analysis of these inputs, performed in the next step, enabled us to exclude the documents, which did not present any malicious behavior.

As outlined in Section 5.1.2, we first deobfuscated our malicious samples with JSDetox [200] and box-js [28]. Still, we could not automate the process due to malicious files conducting environment detection and refusing to execute. As a consequence, each tested sample needed to be, at least *partially*, manually deobfuscated. To reduce the number of files to analyze manually, we clustered our data (by source), based on the syntactic units it contains, using the features from JaSt. From the 122,345 malicious scripts, we got 61 clusters. Subsequently, we randomly selected one file per cluster, deobfuscated and unpacked it until the original payload appeared;[3] i.e., to a stage where no JavaScript was dynamically created (e.g., through means of `eval` or equivalents). In essence, this is the state we assume a malicious entity would have before obfuscation or packing. After deobfuscation, we noticed that eight samples were either benign or incomplete (e.g., we did not have an exploit kit landing page, which prevented us from unpacking the malicious content), and we could not find any valid substitute in the corresponding clusters. In contrast, two files had two malicious behaviors depending on the machine where they were executed. Therefore, they gave us four deobfuscated samples instead of two. Finally, we got 55 working malicious documents. Specifically, we have 30 droppers, which download and execute malware from remote servers, 3 call a PowerShell command, and 2 a VBScript command. We also have 20 different exploit kits (e.g., donxref, meadgive, or RIG), which target vulnerabilities in old versions of Java, Adobe Flash or Adobe Reader plugins, or try to exploit old browsers versions.

To avoid duplicates, we manually iterated over the 55 scripts and looked for similar structures, e.g., the combination of `createElement` and `appendChild` is often semantically equivalent to `document.write`. As mentioned in Section 5.2.3.3, we kept the different syntactic variants (up to three for a given file). In particular, we refer to these variants as *one* seed. This way, for a given malicious behavior (i.e., seed), we can try to find clones with up to three different syntactic variants. Besides, we are working at the AST level; therefore, we consider that two samples with the same AST but a different behavior are identical. After duplicate deletion, we retain 22 malicious seeds to which we added a crypto-miner (without users' consent), as cryptojacking in browsers has become a widespread threat [82, 111, 216]. These 23 unique seeds represent in total 37 different syntactic variants.

Finally, to verify to what extent our dataset is representative of the malicious distribution found in the wild, we extracted 13,884 unique samples from VirusTotal [210]. In particular, we collected them *after* the files we analyzed previously. As before, we clustered them syntactically. We got 8 clusters (cf. Table 5.1), one file of each we deobfuscated. As the 8 deobfuscated samples matched our 23-sample set (7 matched exploit kits and 1 a dropper), we deem our dataset to be saturated.

5.3.1.2 Benign Dataset

Next, we collected benign files, whose ASTs HideNoSeek reproduces. We present our dataset in Table 5.2. Specifically, we statically scraped the start pages of Alexa top 10k websites, also including external scripts. Given the fact that we extracted JavaScript from the start pages of high-profile sites, we assume our samples to be benign.

[3] We discuss possible drawbacks induced by the deobfuscation process in Section 5.4.1

Source	#Samples	#Valid JS
Alexa top 10k	8,673	8,279
Libraries	268	267
Sum	8,941	8,546

Table 5.2: Benign JavaScript dataset

At the same time, we downloaded the most popular JavaScript libraries, according to W3Techs [211].

5.3.2 Evasive Sample Generation

HideNoSeek leverages our 23 malicious seeds to generate malicious JavaScript samples whose ASTs match existing benign ones. We first report on the samples our system could craft per malicious seed before evaluating the impact our attack would have on the highest-ranked web pages and libraries.

5.3.2.1 Analysis per Malicious Seed

In our first experiment, we study the number of samples that HideNoSeek could produce per malicious seed, i.e., the number of ASTs of Alexa top 10k websites that HideNoSeek could reproduce per seed. During the deobfuscation phase presented in Section 5.3.1.1, we noticed that exploit kits from the same family (based on anti-virus labels) could have a different syntactic structure as well as a different behavior. In these cases, they appear several times in the seeds from Table 5.3. This table indicates the number of malicious samples crafted per malicious seed (#Samples), the number of nodes that HideNoSeek had to adjust due to the replacement of benign sub-ASTs with syntactically equivalent malicious ones (#Adjustments), as well as the average number of nodes contained in the crafted samples (#Nodes)–i.e., the average number of nodes in the benign samples whose ASTs a given malicious seed reproduces. In particular, we make a distinction between the samples crafted with the benign AST of a top 1k website compared to a top 10k one. In practice, HideNoSeek could leverage 22/23 seeds to generate 91,020 malicious samples, which reproduce ASTs of Alexa top 10k websites. More specifically, it could rewrite the ASTs of 9,725 top 1k websites, for malicious purposes. In fact, the number of crafted samples is not linear, and, proportionally, we tend to produce more samples for the first 1,000 websites. As an example, for *Blackhole1* we could have expected to generate around 5,600 samples reproducing ASTs of Alexa top 10k, but in practice we got 10% less; for *Crypto-miner* we even got 65% less than expected. Still, the start pages of the 1,000 most consulted websites do not seem to be larger (in terms of delivered JavaScript) than the start pages of the top 10k. It is rather the opposite since, on average, our ASTs contain more nodes for pages from Alexa top 10k than for Alexa top 1k. Nevertheless, the first 1,000 websites seem to have a more complicated structure with, in particular, more data flows: for each replacement HideNoSeek made, it had to adjust more nodes for Alexa top 1k than for the top 10k.

Seed	Alexa top 1k			Alexa top 10k		
	#Samples	#Adjustments	#Nodes	#Samples	#Adjustments	#Nodes
Blackhole1	558	101	106,897	5,040	76	120,189
Blackhole2	558	157	106,897	5,041	100	120,224
Crimepack1	568	45	107,991	5,424	38	117,841
Crimepack2	532	30	113,575	5,072	35	123,490
Crimepack3	127	74	140,399	1,364	97	156,408
Crypto-miner	90	203	80,943	311	119	133,927
Donxref1	629	90	102,710	5,888	36	112,674
Donxref2	454	262	117,727	4,233	221	133,895
Dropper	0	-	-	10	142	153,840
EK	683	27	96,434	6,487	6	104,422
Fallout	427	143	120,639	3,732	67	137,716
Injected1	680	72	97,069	6,447	54	105,217
Injected2	502	131	117,838	4,810	66	128,429
Meadgive	535	55	112,639	5,106	47	122,856
Misc	683	27	96,434	6,487	6	104,431
Neclu1	300	93	122,893	2,890	110	144,388
Neclu2	530	59	113,118	5,031	64	123,925
Packer	204	45	126,836	2,325	48	138,996
PowerShell	507	24	110,891	4,735	22	123,046
RIG1	23	124	179,150	244	171	170,135
RIG2	0	-	-	0	-	-
VBScript1	584	15	100,254	5,275	8	114,502
VBScript2	551	49	105,198	5,068	16	118,696

Table 5.3: Number of samples crafted per malicious seed (i.e., reproducing ASTs of Alexa top 1k vs. 10k websites), average number of nodes that HIDENOSEEK adjusted, and average number of nodes contained in the crafted samples

For this reason, we estimate that the higher complexity of the first 1,000 websites was more favorable to hide our malicious seeds, as their different statements highly depend on each other. In the following, we define the process of *hiding* a malicious seed in a benign sample to refer to the rewriting of the malicious AST into an existing benign one.

The success of our hiding process also depends on the syntactic structures the seeds contain, and to what extent their syntax is identical to benign scripts. With the exploit kit *Misc*, HIDENOSEEK was able to generate evasive samples reproducing ASTs of 78% of the pages from Alexa top 10k. On the contrary, it could craft only 10 samples for the malicious seed *Dropper* and was unable to produce any output for *RIG2*. In fact, these two seeds contain syntactic structures, which are practically never present in benign documents. For example, our dropper initially uses three times the construct *new ActiveXObject("object")*, which we could, e.g., map to the benign construct *new RegExp("regexp")*. However, we found no such pattern *three* times in the same benign file. Thus, we looked for a new syntactic construct, semantically equivalent but which could be found in benign documents too. For this purpose, we investigated the most common syntactic structures that are shared between our malicious seeds and benign dataset. We found the following structure *a.b("")* in 118,700 statements that matched our benign and malicious documents. As a consequence, we replaced the previous malicious dropper construct with its equivalent *WScript.CreateObject("object")* and could this time hide the malicious seed in ten benign documents. Nevertheless, our tool crafted 4,735 samples for the *PowerShell* seed, which is a dropper too. Therefore, an attacker could still hide a dropper in 4,735 web pages from Alexa top 10k. As for *RIG2*, it contains

Rank	Domain	#Seeds	#Nodes
1	google.com	18	58,322
2	youtube.com	18	151,527
3	facebook.com	13	143,772
4	baidu.com	8	35,018
5	wikipedia.org	0	90
6	qq.com	14	54,450
7	yahoo.com	14	67,264
8	taobao.com	19	89,910
9	tmall.com	17	63,102
10	amazon.com	17	36,060

Table 5.4: Number of samples crafted per Alexa top 10 domain and number of nodes contained in the crafted samples

complex syntactic structures, such as *window.frames[0].document.body.innerHTML*, that benign web pages might tend to simplify by, e.g., storing this statement into several variables.

Overall, and out of the 23 malicious seeds HIDENOSEEK got as input, it was able to produce malicious samples for 22 of them. In total, it generated 9,725 malicious samples, which reproduce benign ASTs of Alexa top 1k websites and crafted 91,020 samples for the top 10k. Still, we believe that we could produce even more evasive samples by using different syntactic structures for the seeds. As shown previously, we can identify the most common patterns between the benign and malicious datasets; thus, we envision that malware authors could adjust their code to favor popular constructs that have a high chance of being used by benign samples too.

5.3.2.2 Impact of the Attack

By design, HIDENOSEEK can rewrite the AST of a given malicious seed to reproduce ASTs of different web pages, thereby misleading static detectors. Next, we study the impact our attack would have. For this purpose, we target specific domains and focus, in particular, on hiding malicious JavaScript in the most frequented web pages and libraries. Table 5.4 indicates how many malicious seeds HIDENOSEEK could hide in Alexa top 10, i.e., with how many seeds it could reproduce the benign AST of a given domain. Specifically, we hid 78% of our seeds in the start pages of the two most visited websites, which would maximize the impact of our attack in terms of infected users. Except for *wikipedia.org*, where HIDENOSEEK did not report any clone,[4] we hid on average more than 61% of our seeds in the remaining top 10 websites. We naturally know that for the attack to be effective in practice, the server of these websites should have been compromised so that attackers could replace an original web page with a HIDENOSEEK crafted one. Should that happen, the modified website version would

[4]The start page of *wikipedia.org* contained almost no JavaScript code. For this reason, its AST only had 90 nodes, which is, e.g., 648 times less than *google.com*, and prevented HIDENOSEEK from finding any match with malicious seeds

Library	Alexa usage (%)	Most common version	#Seeds	#Nodes
jQuery	73.5	1.12.4	17	35,511
Bootstrap	18.1	3.3.7	12	10,973
Modernizr	11.4	2.8.3	5	3,174
MooTools	2.4	1.6.0	15	27,786
Angular	0.4	1.7.5-min	17	60,234

Table 5.5: Number of samples crafted per popular JavaScript library and number of nodes contained in the crafted samples

be harder to spot than, e.g., the British Airways attack [107], because of its structure exactly reproducing an existing benign syntax.

A second way of infecting web pages consists in infecting the libraries they use. In this experiment, we consider five of the most popular JavaScript libraries, based on the proportion of websites using them [211], and study the number of malicious seeds we could hide inside. As presented in Table 5.5, we hid between 22% and 74% of our seeds in these popular libraries, i.e., we could reproduce the ASTs of the libraries between 5 and 17 times, based on our 23 seeds. While we could hide on average 14 seeds in each Alexa top 10 website, we can reproduce on average the ASTs of each library 13 times. Similarly to Android malware in repackaged applications [20, 180, 233], HIDENOSEEK could alter benign libraries and present them as an improved version of the original one, for malicious purpose. In particular, such a modification of jQuery 1.12.4 would affect 29.7% of the websites [211] which underlines the impact our attack could have.

5.3.3 Validity Verifications

Based on the insights that HIDENOSEEK could leverage 22 out of 23 seeds to craft 91,020 samples, which have the same AST as scripts extracted from Alexa top 10k websites, in this section, we verify the validity of the samples we produce. After highlighting the fact that a crafted sample has the same AST as the original benign instance, we showcase that they also share most tokens. Finally, we analyze the proportion of samples crafted from jQuery that still run and verify that their malicious semantics can be triggered.

5.3.3.1 Crafted vs. Benign Samples: AST Comparison

First and by construction, all samples crafted by HIDENOSEEK have the same AST as the benign scripts used for the replacement process. Without further testing, this guarantees that malicious JavaScript detectors purely based on syntactic features (e.g., JAST or JSTAP's `ngrams` AST module) are not able to distinguish them.

5.3.3.2 Crafted vs. Benign Samples: Token Comparison

Second, most of the tokens are identical between an initial benign file and a crafted one. The minor differences may come from `Literal` nodes, which can be translated into several tokens, as highlighted in Section 5.2.3.3. On average, 0.29 tokens differ for each 91,020 file crafted from Alexa top 10k websites (containing on average 127,693

nodes). Depending on the targeted detector, this difference could be sufficient to prevent the evasion. In practice, though, we showcase in Section 5.4 that the lexical detector CUJO is highly affected by HIDENOSEEK. In fact, we would produce, on average, at most three samples with one token differing from the original benign inputs every ten crafted script. Thus, the majority of our generated samples also reproduces the original lexical units.

5.3.3.3 Crafted Samples Still Running?

Third, HIDENOSEEK rewrites the ASTs of malicious JavaScript inputs into existing benign ones. To this end, it replaces benign sub-ASTs with syntactically equivalent malicious ones, which could result in the crafted samples not running anymore. To decrease the proportion of broken samples, we implemented a module, which adjusts the nodes impacted by our replacement process by following the data flow edges originating from the benign nodes we replaced (cf. Section 5.2.4.2). Still, some adjustments may not be working in the context where they have been transplanted, e.g., trying to get the length of an `undefined` object will throw an error. In addition, HIDENOSEEK searches for clones between a benign and a malicious input. Nevertheless, it is also valid to replace two independent benign sub-ASTs with two syntactically equivalent malicious ones, which include variables declared in the global scope that depend on one another. In this case, we have to ensure that the execution order of the variables is respected to avoid a `ReferenceError` at run-time.

To verify the correct execution of our crafted samples, we used the library *jsdom* [95]– which emulates web browser functionalities, e.g., DOM elements–to test, with *Node.js*, JavaScript implementations using Web standards. This way, we can ensure that we did not break functionality that requires DOM components. However, this environment cannot be used to test scripts extracted from websites, as *jsdom* is a placeholder, and the downloaded JavaScript often relies on specific constructs in the DOM. Therefore, we executed every crafted sample from standalone benign libraries, like jQuery, to verify if they could still run without throwing, e.g., a `ReferenceError`. Out of the 1,631 samples we crafted from jQuery, 1,175 were still running (72%).

In addition, we could hide 19/23 malicious seeds in jQuery libraries.[5] With the exception of *Crimepack3* (accounting for two jQuery crafted files), each seed has at least one running sample. Also, our approach is more oriented toward the impact our attack would have with the working crafted samples and less toward the proportion of working samples HIDENOSEEK generates. For this purpose, related works [51, 129, 189, 228] combined their implementation with an oracle, which dynamically tested the validity and maliciousness of the samples they produced. In our specific case, such an infrastructure could not be built given the complexity of emulating environments specific to each web page that should have been tested. Still, we were able to leverage 19 malicious seeds to produce 1,175 working versions of jQuery, which have the same AST as the original ones.

[5] In Section 5.3.2.2, we could hide 17 seeds in jQuery version 1.12.4 specifically, while we consider all different jQuery versions in this section

5.3.3.4 Crafted Samples with a Malicious Behavior?

Finally, besides verifying if we could execute the samples crafted by HideNoSeek, we checked their maliciousness. To this end, we randomly selected 5 working crafted samples per malicious seed. In particular, we showed previously that 18/19 seeds generated running samples, based on jQuery. As one seed only has 3 working samples, we tested 88 crafted samples. We analyzed them in two dimensions.

First, we executed these 88 crafted samples in a web browser and manually verified which malicious statements were already called during initialization and which required manual calling. In particular, the jQuery library defines objects and methods for future use and does not necessarily directly call them. As a consequence, we searched for all malicious parts in the considered samples and instantiated, e.g., the functions or objects (with correct parameter values) required to execute the malicious functionalities. Still, if the malicious part was defined, e.g., within a closure, we could not call it. In addition, if the considered malicious element was in an `if` condition, for the sake of simplicity, we first changed the condition to *true*. Out of the 88 samples we analyzed, we validated the correct execution and the expected malicious behavior of 59 (67%).

Second, changing `if` conditions to *true* is not acceptable, as it changes the AST of the crafted samples, while our attack aims at reproducing existing benign ASTs. Still, given our attacker model, malicious actors can freely modify the code as long as they do not change the AST. For this reason, they can either directly manipulate the conditions that have to evaluate to *true* for their attack to be effective or change the environment prior to the conditions (e.g., change the value of a variable defined *before* the condition and impacting its outcome), with respect to the AST. To semi-automate this process, HideNoSeek includes a module, which slices backward along the control flow edges of a replaced node and outputs the different conditional statements that need to evaluate to *true* for the malicious code to be executed. In particular, we applied this module to the 59 crafted samples and analyzed their conditional statements. To change them to *true*, without modifying the AST, we followed these five rules:

1. The *and* and *or* operators are both `LogicalExpression` units and can therefore be interchanged.
2. The comparison operators ==, *!*==, and === are all `BinaryExpression` units and can also be interchanged. Still, this is not the case for the operator *!*, which is an `UnaryExpression`. Thus, we need both an element, which directly evaluates to *true* and another one to *false*, hence the following point.
3. Any object whose value is neither `undefined` nor `null` evaluates to *true* [149], e.g., `window` or `String`. These objects can then be combined with the functions, properties, and methods used in Section 5.2.4.2.
4. `undefined` evaluates to *false*, therefore also calling a non-existing property on an object, e.g., *console.foo*. We cannot leverage the `undefined` property to call a method on an object, though. In this case, we can use *window.Boolean(false)*.
5. To access a property of an object, we can use both bracket and dot notations [146], which are syntactically equivalent.

Finally, by combining these rules on the 59 malicious samples, we could modify all the required conditions so that they always evaluate to true.

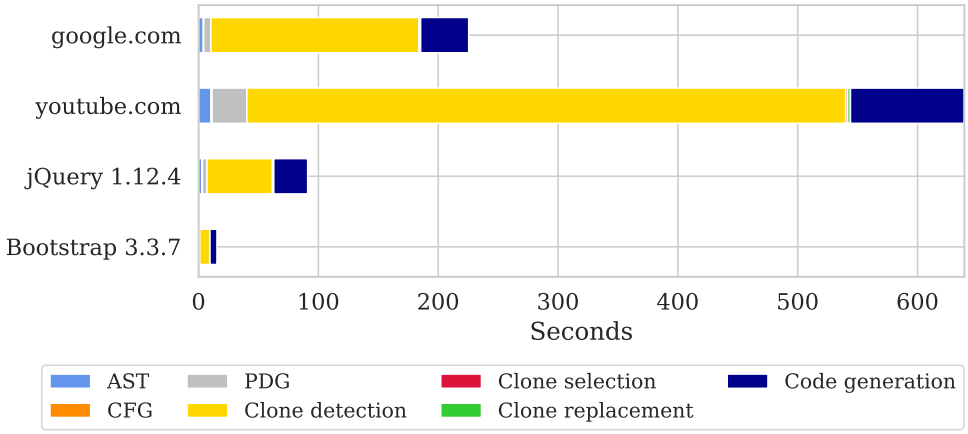


Figure 5.5: Run-time performance of HIDENOSEEK to leverage our 23 malicious seeds to generate samples reproducing ASTs of the two most popular websites and libraries

5.3.4 Run-Time Performance

To evaluate the applicability of HIDENOSEEK in practice, we measured its run-time performance on the two highest ranked Alexa websites (i.e., *google.com* and *youtube.com*) and the two most popular JavaScript libraries (i.e., *jQuery* and *bootstrap*). As indicated in Section 5.3.2.2, we could craft 65 samples, which reproduce the ASTs of the original benign documents.

We took the following measurements from a commodity PC with a quad-core Intel(R) Core(TM) i3-2120 CPU at 3.30 GHz and 8 GB of RAM. Figure 5.5 presents the processing time, for all stages of HIDENOSEEK, to craft 65 samples. The most time-consuming operation corresponds to the clone detection, which is NP-complete and highly depends on the AST size (in terms of nodes). For example, as indicated in Table 5.4, the AST of *youtube.com* contains three times more nodes that *google.com*, and its clone detection phase is also three times longer. Next, generating the code from an AST is also time-consuming, as we traverse the AST of a crafted sample so that Escodegen can produce the code back. Last but not least, the generation of a benign PDG (knowing that we produced the PDGs only once and stored them for future use) may take some time depending on the size of the AST and the complexity of the code (e.g., number of data dependencies).

Overall, the generation of these 65 samples took only sixteen minutes, which underlines the feasibility of our attack.

5.4 Evaluation Against Real-World Detectors

Finally, we evaluate HIDENOSEEK camouflaging of malicious samples on real-world static detectors. We first target traditional structural-based systems (Section 5.4.1) before focusing on the following machine learning-based approaches: CUJO, ZOZZLE, JAST, and JSTAP. We make, in particular, the difference between these classifiers trained with samples from the wild (Section 5.4.2), and those same classifiers also

trained with HideNoSeek samples (Section 5.4.3). Based on the effectiveness of our attack, we ultimately discuss defenses that might prevent our system from crafting samples that will be misclassified (Section 5.4.4).

5.4.1 Evading Structural-Based Detectors

As motivated in Section 5.1.2, we first deobfuscated our malicious files for them to have a more benign-looking syntax (which increased the probability of finding clones in our benign dataset). At the same time, the deobfuscation step implies that we left the malicious logic of our seeds in the open. For this reason, techniques relying on content-matching, e.g., signatures, might be able to detect our crafted samples.

To verify if this is the case, we manually reviewed the Yara rules [231] built for malicious JavaScript detection. These rules are based on strings and specific patterns to describe malware. In particular, the rules targeting exploits, obfuscation, and exploit kits would not work on HideNoSeek crafted samples due to our deobfuscation step. In fact, the considered regular expressions try to match calls to `eval`, `unescape`, special encodings, or specific identifiers (which we renamed during the deobfuscation process) that are not present in the deobfuscated versions anymore. Nevertheless, some rules could still detect JavaScript implementing CVEs, e.g., whenever the reference to a specific instantiated Java object is written in plain text. Yet, in this case, string splitting would foil the pattern-based detection.

In practice, we analyzed the 88 samples from Section 5.3.3.4 with VirusTotal, with between 48 and 60 anti-virus systems. Only 8 crafted samples (5 times for *Donxref1* and 3 times for *PowerShell*) were detected and by at most 2 vendors. Even though the detection accuracy is very low, we envision that HideNoSeek could slightly obfuscate obvious malicious behaviors, e.g., with percent-encoding, to bypass signature-based detection. For example, we slightly obfuscated the *Donxref1* and *PowerShell* seeds. In this case, HideNoSeek could still produce 211 and 795 samples, respectively, but this time, they all evaded VirusTotal detection.

Initially, attackers abused specific code obfuscation techniques to significantly impede the detection and analysis of their malicious code. Still, by doing so, they added specific and recurrent malicious patterns to their JavaScript files. As a consequence, analyses directly based on the code structure were able to leverage such features for an accurate detection. Overall, we can see HideNoSeek as a new form of obfuscation, which, this time, does not add any specific pattern to the samples it modifies. Even though it may adjust some nodes with generic transformations (cf. Section 5.2.4.2), it does not change the code syntax. Implementing a system, which would know HideNoSeek internals and recognize, e.g., function names we changed (such as `toString` or `isFinite`) or integrate a new set of rules targeting deobfuscated malicious JavaScript (such as `ActiveXObject`) is deemed to produce a lot of false positives. We further discuss this assumption in Section 5.4.4.2.

5.4.2 Evading Static Classifiers: Trained from the Wild

Besides evading structural-based detectors, HideNoSeek is also effective against lexical, syntactic, and even control and data flow-based classifiers. In this section, we train

several such detectors with samples found in the wild (i.e., there are no HideNoSeek crafted samples in the training set), and we assess the accuracy of these static systems on our crafted samples.

5.4.2.1 Classifier Training

To verify the evasion capability of HideNoSeek in practice, we leveraged our reimplementation of Cujo and Zozzle (presented in Section 4.2.3.1) as well as our tools JaSt and JStap to classify our crafted samples.

First, we trained these detectors with 10,000 unique malicious files from the wild and as many benign ones. In particular, we randomly picked 8.17% of the malicious samples from each of our malware providers (BSI, Hynek, DNC, and GoS; introduced in Section 5.3.1.1). This way, we built a malicious training set containing 10,000 files, representative of the distribution found in the wild, through our multiple sources and the respect of their initial proportions (cf. Table 5.1). For the benign part, we randomly selected 5,000 unique benign samples from Alexa and 5,000 from Microsoft products (Microsoft Exchange 2016 and Microsoft Team Foundation Server 2017) to combine both web and non-web JavaScript. We deem our model to be balanced, with as many malicious as benign samples. To improve the experiment's reproducibility and limit statistical effects from randomized datasets, we repeated the previous procedure five times and averaged the detection results.

5.4.2.2 Evasion in Practice

In practice, we leveraged the previously trained detectors to classify the 118,052 samples[6] that HideNoSeek generated. Table 5.6 sums up our main findings in terms of the number of HideNoSeek crafted samples correctly recognized as malicious (#TP) and the resulting false-negative rate (FNR). In addition, for the samples previously accurately recognized as malicious, i.e., true positives, we extracted the original benign inputs used for the camouflage (we report on their number, #Samples) and classified them with the same detectors as previously, to report on the false positives (#FP) too.

- **JaSt:** As expected and by construction, our attack foils JaSt, which purely relies on the AST for malicious JavaScript detection. In particular, JaSt misclassifies 99.98% of the samples. Still, it could recognize 26.6 crafted samples as malicious (accuracy: 0.02%). Since these files reproduce existing benign ASTs, we extracted the original 6.2 benign inputs (on average) used to produce these 26.6 samples and classified them with JaSt. All of them are naturally false positives, since they have the same AST as inputs recognized as malicious. We obtain similar results with the `ngrams` AST module of JStap, as it also directly leverages the AST for a malicious JavaScript detection (FNR: 99.98%).
- **Zozzle:** On the contrary, Zozzle also includes the text of AST nodes as an additional feature. Nevertheless, it is not sufficient to enable an accurate detection, as Zozzle could detect only 3.8 crafted samples (accuracy: 3.2E-3%), which used 0.8 benign files for the hiding process. As none of them were classified as malicious (i.e., no false

[6] We classify all crafted samples, i.e., also produced by variants from the same seed

Detector	HideNoSeek		Alexa considered	
	#TP	FNR (%)	#Samples	#FP
JaSt	26.6	99.98	6.2	6.2
Zozzle	3.8	100.00	0.8	0.0
Cujo	15.8	99.99	7.6	6.6
JStap: `ngrams` PDG	28.4	99.98	3.2	2.6
JStap: `value` PDG	24.4	99.98	2.0	2.0

Table 5.6: Classification of HIDENOSEEK samples: detectors trained from the wild

positives), 0.8 scripts changed classification between the benign and the malicious variants (which is naturally too low to enable an accurate detection of our samples). Thereby, HIDENOSEEK can also evade syntactic detectors leveraging node value information as additional features.

- **Cujo:** Similarly, CUJO detected 15.8 crafted samples (accuracy: 0.01%), which reproduce the AST of 7.6 benign files (on average). Since 6.6 of them are false positives, 1 script changed classification between the benign and the malicious versions. Thus, even though some tokens may differ between the two file versions, the slight token differences have a negligible impact on the overall sample classification.
- **JStap's PDG-based modules:** Next, we target control and data flow-based detectors. While our attack mostly kept the control flow structure of our benign files (mainly due to our clone replacement process), we may observe some differences at the data flow level. In fact, if we replace two benign statement nodes independent of one another with two malicious statement nodes depending on each other, the resulting crafted sample would have a data flow that was not originally present in the benign file. To determine if this may be sufficient to evade detection, we classified our crafted samples with the JSTAP modules working at the PDG level. For the `ngrams` variant, we could detect 28.4 samples as malicious, leading to 99.98% misclassifications and only 0.6 classification changes between benign and malicious versions. Similarly, the `value` module has a false-negative rate of 99.98% and no classification changes. We observed similar results for the purely control or data flow-based modules with between 99.95% (`ngrams` CFG) and 99.98% (`ngrams` DFG) misclassifications, meaning that there are few differences at control and data flow levels between benign and crafted samples. Therefore, HIDENOSEEK can also evade detectors leveraging control and/or data flow information to detect malicious JavaScript samples.
- **JStap's pre-filters:** Finally, we leveraged the pre-filtering pipelines of JSTAP to classify our crafted samples. We report on our findings in Table 5.7. As described in Section 4.3.1.3, in a first pre-filtering step, we classified only the samples for which the `ngrams` AST, `value` tokens, and `value` PDG modules made the same predictions. In fact, we have shown that this combination is a way to have extremely reliable predictions on a subset of our samples. With HIDENOSEEK, though, 118,003.4 samples on average have unanimous predictions, but they are all classified as benign, leading to a false-negative rate of 100% (and to no benign input considered for further analysis as no malicious file was detected). In a second pre-filtering step (cf. Section 4.3.2), we classified the remaining 48.6 samples with the unanimous predictions (if any) of the

	HideNoSeek			Alexa considered	
	#Unanimous	**#TP**	**FNR (%)**	**#Unanimous**	**#FP**
Pre-filter 1	118,003.4	0	100	-	-
Pre-filter 2	23.2	0	100	-	-

Table 5.7: Classification of HIDENOSEEK samples: JSTAP's pre-filters trained from the wild

`ngrams` tokens, `ngrams` AST, and `value` AST modules. As previously, they made the same predictions only to label samples as benign. All in all, JSTAP's pre-filtering pipelines did not enable to classify a single input correctly, due to the combined detectors only making the same predictions to classify the crafted samples as benign (only the remaining 25.4 samples, on average, that had different predictions, may be recognized as malicious, after more costly manual or dynamic analyses).

5.4.2.3 Summary

To sum up, and as expected, HIDENOSEEK foils detectors directly and solely leveraging the AST structure to distinguish benign from malicious JavaScript inputs. In addition, we showcased that systems using node value information, tokens, control, and/or data flow features are also vulnerable to our attack. In fact, reproducing an AST also reproduces most of the previous features, thereby evades the corresponding detectors. Finally, we combined the predictions of several classifiers, which were unanimous in misclassifying our crafted samples as benign. Overall, the tested static detectors, trained with samples from the wild, all have a false-negative rate between 99.95% and 100% on our HIDENOSEEK crafted samples, which highlights the effectiveness of our attack.

5.4.3 Evading Static Classifiers: Retrained with HIDENOSEEK Samples

Still, the considered classifiers all combine static features with machine learning algorithms to distinguish benign from malicious inputs. In this section, we now leverage the fact that these detectors are able to *learn*. In particular, we retrained the previous classifiers with samples found in the wild and samples crafted by HIDENOSEEK so that our attack is not *unknown* anymore. As previously, we then assess the accuracy of the detectors on our crafted samples and, for the true positives, on the benign files used for the camouflage.

5.4.3.1 Classifier Training

To verify the effectiveness of our attack when the targeted detectors are aware of HIDENOSEEK, we retrained these classifiers by also including some crafted samples in our five training sets. Specifically, we randomly selected 5,000 malicious files from our previous training sets (cf. Section 5.4.2.1) and randomly added 5,000 HIDENOSEEK samples to illustrate the adaptation of the targeted detectors to our attack. We kept the same benign sets as previously. This way, we built five new training sets containing each time 10,000 malicious and as many benign samples. To avoid any bias on the detectors'

Detector	HideNoSeek		Alexa considered		
	#TP	FNR (%)	#Samples	#FP	FPR (%)
JaSt	108,164.4	4.32	5,907.8	5,907.8	100.00
Zozzle	103,513.4	8.44	5,695.0	5,054.0	88.74
Cujo	108,489.0	4.04	5,941.6	5,564.6	93.65
JStap: `ngrams` PDG	106,911.4	5.42	5,781.2	5,757.6	99.59
JStap: `value` PDG	107,023.8	5.32	5,888.0	5,665.0	96.21

Table 5.8: Classification of HIDENOSEEK samples: detectors retrained with HIDENOSEEK samples

accuracy, we excluded the HIDENOSEEK samples we used to train the classifiers from their corresponding test sets. Therefore, in the following, we present our new detection results on the remaining 113,052 HIDENOSEEK samples.

5.4.3.2 Evasion in Practice

Contrary to the previous experiment, where the detectors had never seen HIDENOSEEK samples before and misclassified nearly all of them, most of our crafted samples are now correctly labeled as malicious. Table 5.8 sums up our main findings.

- **JaSt:** Specifically, JAST was this time able to detect 108,164.4 crafted samples on average, meaning that it has a false-negative rate of only 4.32% on such inputs (compared to 99.98% previously). Nevertheless, the 5,907.8 benign files used to generate the true positives are now *all* misclassified as malicious, hence a false-positive rate of 100% on these samples. In fact, we trained JAST with malicious inputs, which have the same AST as existing benign samples. Therefore, JAST could not learn any feature specific to one class and had to decide whether to classify all such files as malicious (thereby, the benign samples would be false positives, actual case) or as benign (therefore, HIDENOSEEK crafted samples would be false negatives). This way, most of our crafted samples are now detected but at the cost of the benign inputs, which are misclassified as malicious.
- **Zozzle and Cujo:** We observe a similar trend for ZOZZLE and CUJO, which accurately detected 103,513.4 and 108,489 crafted samples, respectively. Thus, the corresponding false-negative rates are 8.44% and 4.04% (compared to over 99.99% in the previous experiment). For ZOZZLE, 5,695 benign inputs led to the true positives, 5,054 of which it misclassified as malicious, meaning that it has a false-positive rate of 88.74%. It also means that 641 samples changed classification between the benign and malicious variants, which represents only 11.26% of our benign set. Similarly, CUJO has a false-positive rate of 93.65% on the benign inputs used for the camouflage, and CUJO reported 377 classification changes. For both tools, even if the proportion of classification changes is higher than in Section 5.4.2, it is still too low to provide a reliable detection mechanism against our attack. Therefore, considering node value information from the AST or token-based features (including type information) does not enable to accurately detect HIDENOSEEK samples.

	HideNoSeek			Alexa considered	
	#Unanimous	**#TP**	**FNR (%)**	**#Unanimous**	**#FP**
Pre-filter 1	108,783.2	106,101.2	2.47	83.2	83.2
Pre-filter 2	1,452.8	190.6	86.88	18.4	18.4

Table 5.9: Classification of HIDENOSEEK samples: JSTAP's pre-filters retrained with HIDENOSEEK samples

- **JStap's PDG-based modules:** Next, we classified our crafted samples with the PDG-based modules from JSTAP. As for JAST, ZOZZLE, and CUJO, most of our malicious samples are correctly detected, as indicated by our low false-negative rates of 5.42% (`ngrams` variant) and 5.32% (`value`). Still, and as previously, these correct predictions occur at the cost of the benign samples, between 96.21% and 99.59% of which are misclassified as malicious. We note, in particular, that 23.6 (`ngrams`) and 223 (`value`) samples changed classification between the benign and malicious versions. While it is again too low to be leveraged as a defense mechanism, we observe that having node value information, similarly to ZOZZLE, contributes to these classification changes. Still, too many features stay identical between the original benign inputs and our crafted samples, thereby leading to such high false-positive rates on the benign samples used for the camouflage. The results are similar for the purely control or data flow-based detectors, with a false-negative rate between 5.27% (`ngrams DFG`) and 5.83% (`ngrams CFG`) on HIDENOSEEK samples and a false-positive rate between 96.23% (`value CFG`) and 100% (`ngrams CFG`) on the corresponding benign inputs.
- **JStap's pre-filters:** Finally, and similarly to Section 5.4.2, we applied JSTAP's two pre-filtering layers to classify our crafted samples. As presented in Table 5.9, our first pre-filter could unanimously classify 96% of our crafted sample set with a false-negative rate of only 2.47%. Still, it misclassified all the 83.2 benign inputs for which the three detectors made the same predictions. Also, it would send almost 6,000 benign samples to more costly manual or dynamic components because of the absence of unanimous predictions. In a second pre-filtering step, we classified the remaining 4,268.8 malicious samples, which had conflicting labels previously. This time, we could classify only 34% of them unanimously (standing for a remaining 1.3% of our initial malicious set) and with a high false-negative rate of 86.88%. In addition, all the benign samples with unanimous predictions got misclassified. Overall, combining JSTAP modules enabled us to slightly increase the true-positive rate on HIDENOSEEK samples. In turn, all benign inputs with unanimous predictions were misclassified, meaning that our pre-filtering pipelines also cannot accurately classify our crafted samples.

5.4.3.3 Summary

To sum up, the targeted machine learning-based detectors are unable to learn from HIDENOSEEK samples. When they are not aware of our attack, they misclassify our samples over 99.95% of the time (cf. Section 5.4.2). On the contrary, after having

been trained with such samples, they can recognize HIDENOSEEK inputs with a true-positive rate of 95.96%, in the best case[7] but at the cost of the benign files used for the camouflage, which they misclassify between 88.74% and 100% of the time. This way, retraining the classifiers is not a defense against HIDENOSEEK, which still foils the lexical, syntactic, control, and/or data flow-based detectors we considered, even when combined.

5.4.4 Potential Detection Strategies

We highlighted and quantified previously the effectiveness of our attack in terms of sample misclassifications. In this section, we present defenses against attacks on machine learning systems and argue why they would not work for HIDENOSEEK. We finally discuss some strategies that might enable to detect our crafted samples.

5.4.4.1 Existing Defenses

The field of attacks against systems using machine learning for classification purposes is vast, e.g., in the malware (cf. Section 5.1.1) or image domains [69, 160]. Different studies assessed the security of learning-based detection techniques by evaluating the hardness of evasion, according to the information leaks an attacker might have, such as black-box access to a targeted classifier or dataset related insights [18, 30, 31, 52, 62, 189]. More recently, systems have been proposed to detect adversarial examples–i.e., inputs specifically crafted to foil a target classifier. They rely on the detection of unreliable results [184], statistical tests [71], dimensionality reduction [17, 227], the detection of adversarial perturbations [131, 133], vectorization [97], or differential privacy [119]. Nevertheless, we envision that none of them would work for our attack, as we perfectly reproduce an existing benign syntax instead of merely injecting benign features.

5.4.4.2 Mitigation Strategies

Nevertheless, HIDENOSEEK adjusts the code it crafted with calls to, e.g., `toString` or `isFinite`, which can create seemingly dead code (due to the lack of proper usage of the result) whose frequency could be an indicator of our crafted samples. Still, relying on such artifacts is likely to produce a lot of false positives. Specifically, we showcased in Section 5.4 that the classifiers we considered were not able to learn from such patterns. In addition, new detectors focusing solely on such features are likely to fail as also evidenced by, e.g., Google sites making extensive use of parameterless `toString` invocations [170]. Similarly, given the quality of JavaScript code in the wild, our experience leads us to believe that otherwise useless variables are not a good indicator of HIDENOSEEK crafted samples either: for example, when calls to *console.log* are commented out without removing the assignments of variables only used in that call.

Apart from this, another possibility to detect HIDENOSEEK samples would be to (a) recognize the original benign input used for the camouflage attack, and (b) notice that it differs from the benign file it is supposed to be. If the original sample is recognized, a checksum test should indicate whether it is the original version or not. Still, official

[7] 97.53% for JSTAP's first pre-filtering layer, on 96% of our crafted sample set

library source code can be altered for benign purposes, like functionality extension, caching proxies, or stored together with other libraries. In particular, we leveraged *retire.js* [156] to detect the different versions of jQuery used by Alexa top 10k websites. In practice, none of the 73 versions matched the hash given on the official jQuery web page. In fact, they were either combined with other JavaScript code or contained, e.g., the name of the caching proxy, essentially nullifying a checksum. Therefore, hash testing is not a reliable solution against our attack either.

Last but not least, HideNoSeek is an attack against *static* malicious JavaScript detectors. Therefore, it does not necessarily foil hybrid or dynamic systems such as Rozzle [108], J-Force [105], or JSForce [85], which force the JavaScript execution engine to test and analyze all execution paths systematically. Similarly, Kapravelos et al. could detect with *Revolver* [103] that an original benign and a crafted sample have the same AST, but the dynamic part of *Revolver* would classify the samples differently, hence underlining the evasion attempt. We contacted the authors to test our crafted samples with *Revolver*, but the system is not available anymore, which prevented this experiment.

Nevertheless, dynamic detectors are usually slow, thus rather work on pre-filtered lists of samples likely to be malicious [27]. In practice, such lists are rather generated by static systems, e.g., lexical or syntactic detectors, which are faster and (generally) accurate. Still, we showcased in Section 5.4.2 that, when the classifiers have no knowledge about our attack, they misclassify our crafted samples as benign. Therefore, in this case, our HideNoSeek samples may not even be dynamically analyzed.

5.5 Summary

In this chapter, we proposed HideNoSeek, our generic camouflage attack, which automatically rewrites the ASTs of malicious JavaScript inputs to reproduce existing benign ones (*RQ3*). To generate an evasive sample, HideNoSeek builds the AST of a benign and the AST of a malicious input, both of which it enhances with control and data flows before looking for syntactic clones. If HideNoSeek can find all the malicious syntactic structures in the considered benign AST, it replaces the benign clones with their malicious syntactic equivalents. Then, HideNoSeek leverages the benign data flows to adjust the remaining code, with respect to the original benign AST, to retain the malicious semantics. By construction, our attack evades detectors directly and solely relying on the AST for malicious JavaScript detection, e.g., JaSt. In addition, we showcased that HideNoSeek is also effective against lexical, syntactic, control and/or data flow-based classifiers, even when they are combined, e.g., Cujo, Zozzle, JStap and its pre-filters. In fact, reproducing benign ASTs also reproduces most of the benign features leveraged by such detectors. In practice, when these classifiers have never seen HideNoSeek samples before, they have a false-negative rate between 99.95% and 100% on such inputs. On the contrary, when the classifiers are aware of our attack and have been trained with our samples, they may be able to accurately detect most of them but, in turn, misclassify between 88.74% and 100% of the benign inputs used for the camouflage. As a consequence, most of the static detectors relying on, e.g., lexical, syntactic, control, and/or data flow features, are impacted by our attack, thus are inept to handle HideNoSeek samples.

This chapter concludes the research work we conducted in this thesis about malicious JavaScript, both in terms of detection and evasion. Still, the different static code abstraction and analysis techniques, which we implemented and leveraged previously, can be extended to analyze browser extensions. In particular, the following chapter is orthogonal to our previous contributions, insofar as we do not focus on deliberately malicious code anymore but on suspicious data flows in extensions, which may lead to security- and privacy-critical issues.

6

DOUBLEX: Statically Analyzing Browser Extensions at Scale

In the three previous chapters, we focussed on JavaScript samples that are inherently *benign* or *malicious*. Specifically, we presented JaSt and JStap to automatically distinguish benign from malicious JavaScript instances. Then, we showed with HideNoSeek that we could automatically rewrite malicious JavaScript files into having the same syntactic structure as existing benign samples, thus misleading the considered detectors. Still, besides malicious documents, attackers can also leverage *benign-but-buggy* code to perform malicious activities.

Due to their high privileges, browser extensions are a target of choice for malicious actors. In practice, web pages and extensions may be isolated, but they can still communicate through messages, which may be controlled by attackers. This chapter thus revolves around *RQ4: To what extent and how can we statically analyze browser extensions to detect suspicious data flows from and toward security- and privacy-critical APIs?* To answer the previous question, we introduce our static analyzer DoubleX. DoubleX abstracts extension code with a graph, including control and data flows, pointer analysis, and models the message interactions within and outside of an extension. This way, we track and detect suspicious flows between external actors and dangerous APIs, as such flows can lead to, e.g., arbitrary code execution in an extension privileged context or sensitive user data exfiltration. In practice, we evaluate DoubleX on 154,484 Chrome extensions and highlight both its precision and recall in terms of vulnerable extensions detected. Finally, we discuss its applicability for an accurate analysis at scale.

6.1 Threat Model

Browser extensions can interact both with web applications and other extensions. By design, malicious actors can send specific messages to a vulnerable extension, tailored to exploit its flaws. In this section, we first discuss the capabilities attackers could gain before motivating and describing the two attacker scenarios we consider.

6.1.1 Attacker Capabilities

By exploiting the privileges of vulnerable extensions, attackers could gain the following capabilities:

- **Code Execution:** attackers can execute arbitrary code in an extension's (or content script's) context. For example, a call to `eval` would enable them to exploit all permissions of a vulnerable extension. Also, through `tabs.executeScript`, they may gain a universal XSS (i.e., the ability to execute code in every website even without a vulnerability in the site itself);
- **Triggering Downloads:** they can download and save arbitrary files on users' machines without prior notice;
- **Cross-Origin Requests:** they can bypass the Same Origin Policy (cf. Section 2.1.2);
- **Data Exfiltration:** they can access sensitive user information such as cookies, browsing history, or most visited sites, leading to, e.g., session hijacking or browser fingerprinting [175, 181, 192, 194].

To gain such capabilities, attackers should be able to influence the input of security-critical sinks or collect the output of privacy-critical APIs in browser extensions. We

Flaw category	All components	High-privilege components
Code Execution	`eval, setInterval, setTimeout`	`tabs.executeScript`
Triggering Downloads		`downloads.download`
Cross-Origin Requests	`$.ajax, jQuery.ajax, fetch, $.get, jQuery.get, $http.get, $.post, $http.post, XMLHttpRequest().open, jQuery.post, XMLHttpRequest.open`	
Data Exfiltration		`bookmarks.getTree, cookies.getAll, history.search, topSites.get`

Table 6.1: Security- and privacy-critical APIs considered depending on an extension components

list the corresponding APIs in Table 6.1. We indicate, in particular, if all extension components can access them or only the high-privilege ones. Also, DoubleX is highly modular and can recognize and analyze other APIs (which did not fit in our attacker models); thus, flag additional suspicious data flows. We give an example in Section 6.2.4.4.

6.1.2 Attacker Models

In this chapter, we focus on two attacker scenarios to exploit the privileges of vulnerable extensions: a *Web Attacker* and an attacker abusing a *Confused Deputy* through an unprivileged extension.

In the first scenario, the attackers can trick a user into visiting a web page that is allowed to communicate with an extension. This page can subsequently send messages to exploit a vulnerable extension. In this chapter, we assume that an extension is susceptible even if only one page is allowed to send messages, i.e., even if only one page can exploit the capabilities of an extension. This is motivated by the fact that WhiteHat Security reported 38% of the applications as having at least an XSS vulnerability in 2018 [204] and even high-profile sites like Google have recently had XSS vulnerabilities [155]. While an XSS in such a high-profile page is in itself troublesome, a compromised extension is more powerful than an XSS in a single page as the Web Attacker can leverage the extension privileges, e.g., to attack entirely unrelated sites.

In the second scenario, the attackers can trick a user into installing a specific extension under their control. As previously, this extension would send the payload to exploit a vulnerable extension (i.e., a confused deputy). A malicious extension using this technique would be harder to detect than a classical malicious one, as it does not need any permission, nor uses any dangerous API, nor makes any suspicious resource access so that its maliciousness stays hidden [25]. The only aim of this malicious extension would be to exploit the privileges of vulnerable ones. As a cover, it could implement innocuous functionality that does not require any privilege, making it easy to pass through the review process [43]. To evaluate the feasibility of an attack through an unprivileged extension, we uploaded such an extension to the Chrome Web Store. Under the pretense of customizing the default new tab page in Chrome, our extension was sending malicious payloads to exploit two vulnerable extensions reported by DoubleX. Our extension was reviewed, and we were notified of its acceptance one day later. Once accepted, we

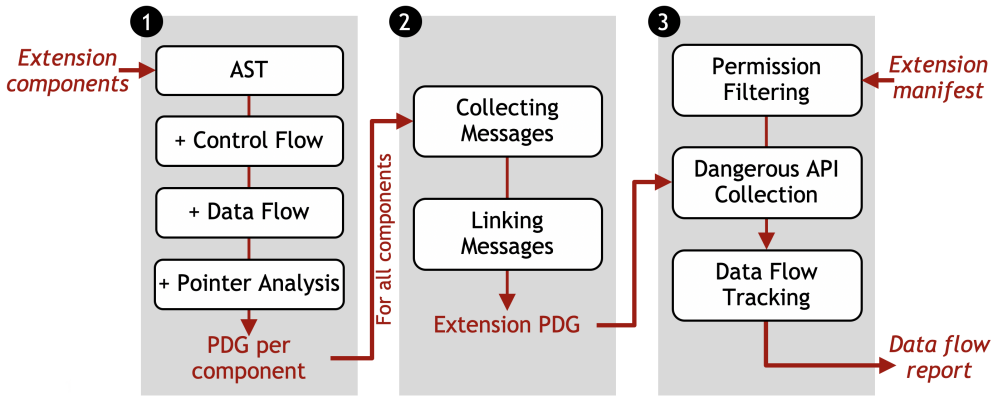


Figure 6.1: Architecture of DOUBLEX

installed the extension along with the two vulnerable ones, and we confirm that we could exploit their vulnerabilities against ourselves. Similarly to CROSSFIRE [25], we would like to stress that our extension was designed as a case study. Specifically, we did not test it against real users, nor harm anyone; we set the extension visibility to *unlisted* (i.e., only people with the link could see it), we did not advertise it, and we confirm that it was downloaded only once (by us, to test it), and then we promptly removed it. Hence, we are confident that neither users were harmed nor details of the vulnerable extensions made public.

In this chapter, we consider that an extension is vulnerable when at least another extension or web page can exploit its privileges to lead to the security or privacy issues we presented.

6.2 DOUBLEX

DOUBLEX performs a fully static data flow analysis of browser extensions to detect those with suspicious external flows. We chose to conduct a static analysis due to its speed and code coverage. This section first provides a high-level overview of our system before presenting its three main components into more details. In particular, we describe how we build the PDG of each extension component independently. Subsequently, we model the interactions between components by leveraging the invoked message-passing APIs; thus, we define the PDG at extension level. We then leverage this graph to detect suspicious data flows.

6.2.1 Conceptual Overview

As illustrated by Figure 6.1, DOUBLEX abstracts the source code of an extension with semantic information and models the interactions within and outside of an extension. This way, we can perform a data flow analysis to identify any path between the attackers and dangerous APIs. In its core, we implemented DOUBLEX in Python. First, we build an AST for each extension component, which we enhance with control and data flow, and pointer analysis information. We refer to the resulting graph as a PDG (Figure 6.1

stage 1). This way, we have a semantic abstraction of extension code, enabling us to reason about conditions leading to specific execution paths and variable dependencies (Section 6.2.2). Next, DoubleX leverages the PDG of each component to build a joint structure representing the extension PDG (Figure 6.1 stage 2). For this purpose, we traverse the PDG of each component independently and collect all message-passing APIs. Based on the component and the API used, we can infer with whom it is exchanging messages. For internal messages, DoubleX links, e.g., the message sent by component A to the message received by component B (for all components) with a *message flow*, to represent the interactions between the components, thus defining the PDG at extension level. In addition, we detect external message-passing APIs to collect both attacker-controllable data and data that could be exfiltrated to the attackers (Section 6.2.3). Finally, we leverage the extension PDG to perform a data flow analysis targeting security- and privacy-critical APIs (Figure 6.1 stage 3). In particular, we look for data flows between the dangerous or sensitive data previously collected and security or privacy-critical APIs. DoubleX summarizes its findings in a fine-grained data flow report (Section 6.2.4).

6.2.2 Generating a PDG per Extension Component

To analyze a browser extension, we first abstract the source code of each component independently. In particular, we model each component with a separate PDG, which includes AST edges, control and data flow edges, and pointer analysis information.

6.2.2.1 Syntactic Analysis

Similarly to Section 4.1.1.2, DoubleX leverages the parser Esprima to generate the AST of each component. Next, to detect whether an extension executes attacker-controllable data or exfiltrates sensitive user information, we need a more complex abstraction of the code that goes beyond its syntactic order. Specifically, DoubleX gives more semantics to the AST nodes by (1) generating and storing their control flows, (2) their data flows, and (3) computing variable values.

6.2.2.2 Control Flow Analysis

DoubleX extends the AST with flows of control to reason about the conditions that should be met for a specific execution path to be taken. To this end, we use our CFG implementation from Section 4.1.1.3. As highlighted previously, Figure 6.2 (considering the blue dotted control flow edges) presents an execution path difference when the `if` condition is true vs. false. Still, the CFG does not enable us to infer whether the condition is true or not.

6.2.2.3 Data Flow Analysis

To reason about variable dependencies and compute variable values, DoubleX adds data flow edges to the CFG, which becomes a PDG. In this chapter, we significantly improve the data flow implementation from Section 4.1.1.4 to tailor it to our extension analysis use case. For example, we add a pointer analysis module to generate the values

```
1 b = 1;
2 if (b === 1) {
3   a = 2;
4 } else {
5   a = 3;
6 }
7 var c = a*a;
```

Listing 6.1: JavaScript code example

of variables. For this reason, we are now considering, e.g., function argument passing (while we previously only needed to know if/where the functions were called). Even though data flow and pointer analyses are interlinked and we perform them in the same CFG traversal, we present them in two sections, for clarity reasons. Also, we chose to build the PDG by traversing the CFG one time (vs. iterating until we reach a fix point), for performance reasons. While it may lead to under-approximations, our analysis stays accurate (cf. Section 6.2.4.4) and is able to scale (cf. Section 6.3). We discuss some drawbacks of our static approach in Section 8.1, though.

Contrary to our previous PDG implementation, to ease the value computation process, we represent data flows between `Identifier` nodes. In particular, we connect `Identifier` nodes (referencing, e.g., variables, functions, or objects) with a directed data flow edge if and only if they are defined or modified at the source node and used (or called) at the destination node, with respect to the scoping rules. If a variable is defined with the `window` object, directly assigned or defined outside of any function, it is in the global scope. Otherwise, the variable can only be accessed in specific parts of the code (the local scope). To keep track of variables currently defined and accessible in a given scope, DoubleX defines a list of *Scope* objects. In particular, we leverage CFG information to build different and independent *Scope* objects to handle variables from branches triggered by exclusive predicates (e.g., a *true* vs. *false* `if` branch), to avoid impossible data flows. When exiting such a conditional node, we merge all variables defined or modified in the different branches to their corresponding scope (i.e., global or specific local scope) to respect the scoping rules and make these variables accessible for future use. This way, DoubleX traverses the CFG and links the encountered variables to their definition or modification sites with a data flow edge. For example, the orange dashed data flow edges in Figure 6.2 represent variable dependencies. Specifically, we link variable b from its definition site (Listing 6.1 line 1) to its usage (line 2). The same applies to a (defined line 3 and used two times line 7).[1]

As far as functions are concerned, we hoist `FunctionDeclaration` nodes at the top of the current scope (as indicated by the ECMAScript specification [56], as they may be first used then defined), and distinguish them from `(Arrow)FunctionExpression` nodes (which have to be defined before usage) [141]. In addition, DoubleX respects the function scoping rules, e.g., closures and lexical scoping. Also, we define a *parameter flow* to link function parameters at the call sites to the definition site, and we keep track

[1] For its definition in line 5 and the absence of a data flow, see the following section

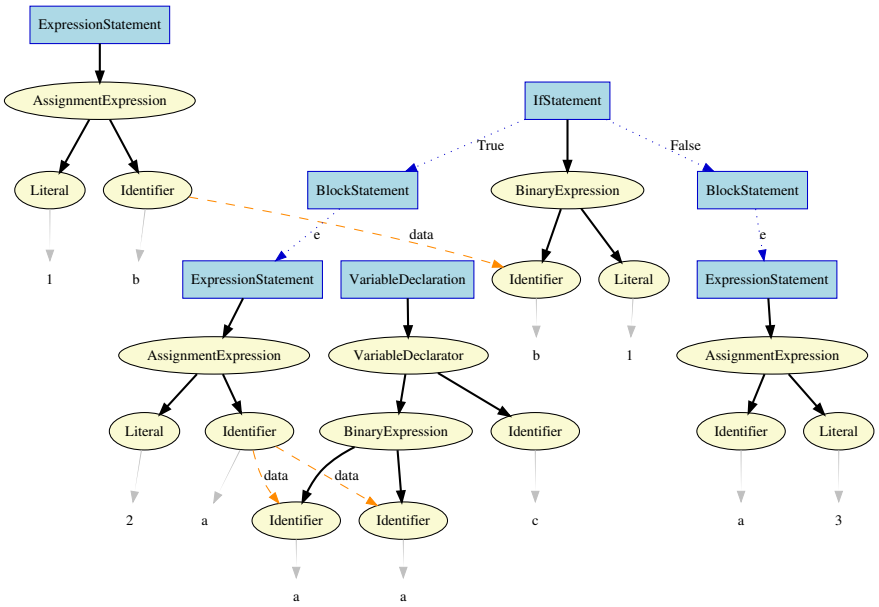


Figure 6.2: AST of Listing 6.1 extended with control & data flows

of the returned values. This way, our analysis is inter-procedural, and we define our PDG at program level.

6.2.2.4 Pointer Analysis

Finally, to compute variable values we follow four main principles:

1. If we already computed a node value, we fetch it from the node `value` attribute.
2. We know the value of a `Literal`, which Esprima stores as a node attribute.
3. Whenever there is a data flow between two `Identifier` nodes (from *source* to *destination*), the *destination* has the same value as the *source*.
4. We define different rules to compute the values of variables, which undergo specific operations. Specifically, we handle assignment, arithmetic, string, comparison, and logical operators.

We illustrate these principles in Algorithm 6.1, which is a simplified extract of our pointer analysis. We call this script at specific nodes while we traverse the graph to perform the data flow analysis.

For example, in Figure 6.2, there is an `AssignmentExpression` (Algorithm 6.1 lines 11-17). Here, the `Identifier` b is declared. To compute its value, we determine the value of its symmetric node (line 15): as the `Literal` value is 1 (lines 3-4), so is b (line 16). Next, there is an `IfStatement`, whose condition is a `BinaryExpression` (lines 18-22). By following the data flow from b backward, we get its value 1 (lines 19 and 5-7). As the condition always evaluates to *true* (lines 20-22), DoubleX solely focuses on this branch (meaning that Listing 6.1 line 5 is never analyzed, hence no data flow from here, which limits over-approximations due to impossible cases). Finally, DoubleX fetches the value 2 from the symmetric node of a and computes the operation

```
   Data: node object n
   Result: n computed value
 1 if n.value is not None then
 2 |   return n.value;
 3 else if n.name == "Literal" then
 4 |   value ← n.attributes["value"];
 5 else if n.name == "Identifier" then
 6 |   if n is the destination of a data flow from source then
 7 |   |   value ← source.value;
 8 |   else
 9 |   |   value ← n.attributes["name"];
10 |   end
11 else if n.name in ("VariableDeclaration", "AssignmentExpression") then
12 |   value ← None;
13 |   find the defined/assigned Identifier nodes; // variable names
14 |   for each Identifier node i do
15 |   |   p ← calculate the symmetric path to i;
   |   |   // + some refinements for Array, Object nodes etc.
16 |   |   i.value ← call compute_value(p)
17 |   end
18 else if n.name == "BinaryExpression" then
19 |   operand1 ← compute_value(n.children[0]);
20 |   operand2 ← compute_value(n.children[1]);
21 |   operator ← n.attributes["operator"];
22 |   value ← operand1 operator operand2;
23 else if ... then
24 |   ...
25 n.value ← value;
26 return value
```

Algorithm 6.1: *compute_value*: computes, sets, and returns a node value

to get the value 4 for *c*. For clarity reasons, we chose a simplistic example. In particular, DoubleX also analyzes asymmetric variable declarations, e.g., in cases of arrays, objects, functions, or `for` loops (which we can therefore simulate), and destructuring assignments. Also, it handles variables defined with `this`, `self`, `window`, and `top` keywords, and recognizes their aliases.

As far as objects and arrays are concerned, we store a handler to their definition site. This way, whenever a specific property is used or modified, we follow the data flows to access the definition site, traverse the corresponding sub-AST to find the property/index, and compute its value. If an object or array is defined on the fly, we store its components in a dictionary, which becomes the handler to the considered object/array. Finally, whenever a function is called, we follow the data flows to find the function definition site.[2] DoubleX then passes the function parameters at the call site(s) to the definition site by leveraging the parameter flows (cf. Section 6.2.2.3) before retraversing the function. For reproducibility purposes, we made our source code available [T4].

[2]Since we hoisted `FunctionDeclaration` nodes at the top of the current scope in Section 6.2.2.3, the function is always first defined, then called

```
1 // Content script code
2 chrome.runtime.sendMessage("Hi BP", function(response) {
3  csReceived = response.farewell; // csReceived = "Bye CS"
4 });
5
6 // Background page code
7 chrome.runtime.onMessage.addListener(
8  function(request, sender, sendResponse) {
9   bpReceived = request; // bpReceived = "Hi BP"
10   sendResponse({farewell: "Bye CS"});
11 });
```

Listing 6.2: Content script and background page communication example

6.2.3 Generating the Extension PDG

In the previous step, DOUBLEX generated the PDG of each extension component independently. Still, to detect suspicious data flows in an extension, we also need to understand the intricate relations between the extension components and detect external messages. To this end, we collect all messages sent and received by each component and order them per message API. This way, DOUBLEX leverages the APIs to know (for a given message) which components are communicating (or if the message is coming from a web page or another extension). In the case of internal messages, we subsequently link the message sent by component A to the message received by component B; thus, defining a PDG at extension level.

6.2.3.1 Collecting Messages

To collect the messages exchanged within and outside of an extension, we traverse the PDG of each component and look for specific messaging APIs. We consider all APIs presented in Section 2.4.3 (both for Chromium-based browsers and Firefox), as well as deprecated APIs, which Chromium still supports, e.g., `chrome.extension.sendMessage`, `chrome.extension/runtime.sendRequest` [36, 38]. Since we compute node values with DOUBLEX pointer analysis module, we can also detect messaging APIs not written in plain text, e.g., string concatenation or referred to over aliases. Once we find a message-passing API, we look for the specific message that is sent (with a distinction between an initial message, `sent`, and a response, `responded`) or received (similarly, getting a message, `received`, and a response, `got-response`). For this purpose, we created an abstract representation of each API (based on the official documentation from Chrome and Mozilla) to know, depending on the number of arguments, which parameter corresponds to the message. For example, in Listing 6.2, the first parameter of `chrome.runtime.sendMessage` is the message sent by the content script, while the second one is a callback to receive the response from the background, UI page, or WAR [37] (the API used also indicates which components are communicating). Once we know the message position, we collect the message. It can either be directly accessible (e.g., "Hi BP" Listing 6.2, line 2), or accessible only after callback resolution, which DOUBLEX performs by following the data and parameter flows (e.g., the callback

	Content script	Background page
message 1	`sent`: *"Hi BP"*	`received`: *request*
message 2	`got-response`: *response*	`responded`: *{farewell: "Bye CS"}*

Table 6.2: Message collection entry for the extension of Listing 6.2 (channel: one-time, deprecated APIs: no)

`sendResponse` is defined Listing 6.2 line 8 and called line 10 with the message as first parameter). In addition, DoubleX analyzes `Promise` [145], such as calls to `.then` or `.resolve`, which Firefox can use instead of callbacks like Chromium. Finally, we store the collected messages. For internal messages, we order them per pair of components, one-time vs. long-lived channels, and deprecated APIs usage or not. Table 6.2 sums up the messages that DoubleX collected. As indicated in Listing 6.2, the content script sends one message and gets a response, while the background page gets a message and responds. For external messages, we keep track of which extension component received (or sent) a message, at which line in the code, and store the corresponding node object for future analyses (cf. Section 6.2.4.3). For example, in Listing 6.3, DoubleX reports the external message *event*, received line 1 in the content script.

6.2.3.2 Linking Messages

Next, for internal messages, we link the messages sent (`sent`, `responded`) by a component to the messages received (`received`, `got-response`) by the other component. This way, DoubleX joins the individual component PDGs with a *message flow*. As represented in Figure A.2 (green solid edges), the message sent by the content script is linked to the message received by the background page, the same applies to the response. Besides, to keep track of variable values, we update the values of the receiver nodes (and those depending on it) with the sender node values. For the example of Table 6.2, we now know the following values: *request = "Hi BP"* and *response = {farewell: "Bye CS"}*. Therefore, we can compute the value of *csReceived* (Listing 6.2 line 3) by leveraging the `ObjectExpression` *{farewell: "Bye CS"}* with the key *farewell*, getting "Bye CS" (cf. pointer analysis from Section 6.2.2.4). Similarly, the value of *bpReceived* (line 9) is *"Hi BP"*. All in all, DoubleX statically produces a graph structure, which gives an abstract semantic meaning to the extension code and models the interactions within and outside of an extension. We refer to this graph as the extension PDG.

6.2.4 Detecting Suspicious Data Flows

Finally, DoubleX leverages the extension PDG to detect and analyze suspicious data flows. First, and for each extension, we prefilter dangerous APIs (i.e., that an attacker could exploit to gain access to an extension's privileged capabilities) based on an extension's permissions. Then, we traverse the extension PDG to collect any prefiltered dangerous APIs. As we flagged external messages in Section 6.2.3.1, we can subsequently perform a data flow analysis (from source to sink) to detect if these critical APIs are

executed with attacker-controllable data or could exfiltrate sensitive user information. Finally, we evaluate DoubleX on a ground-truth dataset.

6.2.4.1 Permission Filtering

Given that an extension cannot be exploited if it does not have the corresponding permissions, DoubleX parses the manifest to automatically generate, for each extension, a list of sensitive APIs that the extension is allowed to access. We list the sensitive APIs we consider in Table 6.1. In practice, for `XMLHttpRequest` and its derivatives (such as `fetch`), we only consider extensions that are allowed to make requests to arbitrary hosts (e.g., through the <all_urls> permission). For the exfiltration APIs, such as `bookmarks.getTree or history.search`, we ensure that the extensions are allowed to access these APIs (i.e., *bookmarks* or *history* permission). Finally, for `tabs.executeScript`, we verify that the extensions can either make requests to arbitrary hosts (similarly to `XMLHttpRequest`) or have the *activeTab* permission [47], which enables to execute code in the active tab without specific *host* permission. In addition, for the code execution APIs (e.g., `eval` or `tabs.executeScript`) in high-privilege components, we verify that the extensions define a Content Security Policy (CSP) [40] that allows their invocations.

6.2.4.2 Dangerous API Collection

To perform its intended functionality, an extension may use specific APIs, which lead to security or privacy issues when attackers can exploit them. The component presenting the vulnerability will determine the attack surface. While background pages and WARs have high privileges, content scripts have lower ones. For this reason, we consider different dangerous APIs, depending on the components we are analyzing (see Table 6.1). As explained in Section 6.2.4.1, we only consider APIs that have relevant permissions.

We detect these APIs by traversing the extension PDG and computing node values. Whenever DoubleX reports a dangerous API, we store the API name, the node object, and its corresponding value for further analyses (cf. following section). Besides, we keep track of the extension component, which uses the dangerous API and the corresponding line number. For example, DoubleX accurately reports the call to `eval` in Listing 6.3, line 2. Even though it is not written in plain text (see the corresponding PDG in Figure A.1), our pointer analysis module computes the correct value. We also report `eval` line 4. Both reports are stored as dangerous APIs for the content script.

6.2.4.3 Data Flow Tracking

After collecting external messages, i.e., messages potentially exchanged with the attackers (cf. Section 6.2.3.1), and detecting the relevant dangerous API invocations (cf. Section 6.2.4.2), DoubleX performs a data flow analysis (from source to sink). Specifically, we aim at finding any path between dangerous or sensitive data and security- or privacy-critical APIs. Based on the way they operate, we distinguish three categories of dangerous APIs (which we refer to as *danger*):

```
addEventListener("message", function(event) {
  window["e" + "v" + "" + "al"](event.data);
  event = {"data": 42};
  eval(event.data);
})
```

Listing 6.3: Vulnerable content script example

```
{"direct-danger1": "eval",
"value": "window.eval(event.data)",
"line": "2 - 2",
"dataflow": true,
"param1": {
    "received": "event",
    "line": "1 - 1"}},
{"direct-danger2": "eval",
"value": "eval(42)",
"line": "4 - 4",
"dataflow": false}
```

Listing 6.4: Extract of the data flow report for Listing 6.3

- **Direct dangers** can directly leverage attacker-controllable data as parameter to perform malicious activities. Such APIs include `downloads.download` and `tabs.executeScript`, knowing that only the high-privilege components can call them (while `eval` can also be used by the content scripts). To handle such APIs, DOUBLEX extracts their parameters so that it can verify if they depend on data coming from the attackers. To limit false positives, we only extract the *relevant* parameters. For example, it is dangerous to have an attacker-controllable input in the second parameter of `tabs.executeScript` (or the first parameter if the tab ID is not indicated), provided it contains the code to be executed [150]. The first parameter, though, only allows to choose the tab to execute a script in, which is worthless for attackers if they cannot control the code.
- **Indirect dangers** work in two steps: first, they have to be called on attacker-controllable data; second, they need to send the results back to the attackers. For example, to perform cross-origin requests, all components can use `fetch` or jQuery APIs such as `ajax` or `post`. We analyze these APIs in two steps: first, we verify if the attackers can control the relevant API parameters and if it is the case; second, we verify if the data sent back to the attackers depends on data the extension received.
- **Exfiltration dangers** directly exfiltrate sensitive user data and do not necessarily need any input from the attackers. Such APIs can only be used by the high-privilege components and include `cookies.getAll`, `bookmarks.getTree`, and `topSites.get`. DOUBLEX extracts the callback parameters of the dangerous APIs and analyzes if they are being sent back to the attackers.

Also, we can handle cases where messages with attacker-controllable data (or data to be exfiltrated) are forwarded back and forth between the extension components before being exploited (as the extension PDG models the interactions between the components).

DoubleX summarizes its findings in a data flow report. Specifically, it indicates the dangerous APIs it found per extension component and danger category (including API name, line number, and computed value), and if it detected a suspicious data flow. When it is the case, it indicates if data was received from or sent back to the attackers (including in which extension component, the line number, API value, etc.).

For example, Listing 6.4 is a simplified extract of the data flow report for the vulnerable content script of Listing 6.3 (the full report is in Listing A.1). In particular, the *value* entry line 2 indicates that DoubleX accurately computes the first call to `eval`, despite string obfuscation. Lines 2-7 show that DoubleX detects a data flow between the first parameter of `eval` line 2 (of Listing 6.3) and the message *event* received line 1 (of Listing 6.3), from a web page. Thus, we report here that the critical API `eval` can be called on attacker-controllable data. The second danger entry (lines 8-11) revolves around the second call to *eval(event.data)* (Listing 6.3 line 4). DoubleX detects that the *event* object has been redefined, as it computes the value 42 for *event.data* (line 9) and labels the danger as not having an externally controllable data flow (line 11). This way, the combination of our data flow and pointer analyses enables us to accurately label the first call to *eval(event.data)* as vulnerable without misclassifying the second one.

6.2.4.4 Evaluation on a Labeled Dataset

To evaluate the accuracy of DoubleX, we rely on a ground-truth dataset of vulnerable extensions identified by Somé [186]. His paper provides a list of extension IDs and corresponding vulnerabilities. Based on the 171 Chrome extensions he reported as vulnerable in 2019, 82 still existed on March 16, 2021, when we collected our extension set (as described in the following Section 6.3.1). We manually analyzed them to verify if they still have the vulnerabilities previously reported. Out of the 82 extensions, 73 are still vulnerable and have a total of 163 vulnerabilities. As Somé considered some APIs that are not part of our attacker models (e.g., `storage`-related APIs[3]), we added them to our dangerous API list (only for this experiment).

DoubleX detects all vulnerabilities for 62 out of 73 extensions, which corresponds to the accurate detection of 151 / 163 flaws (92.64%). For the twelve missing flaws, four are related to dynamic arrays, such as invocations of a function through *handlers[event.message]*, which we cannot statically resolve. For four other cases, the handler function invokes a function that is not defined at this point in the parsing process. While DoubleX correctly hoists `FunctionDeclaration` nodes, this occurs in cases where a function is defined as a variable (i.e., `(Arrow)FunctionExpression` such as *foo = function() {...}*), which should be defined *before* usage, according to the ECMAScript specification [56, 141]. The last four cases are data flow issues related to circular references (i.e., a property within an object accesses another property in the same object).

In addition, for six extensions, which have not been updated since Somé's analysis, we report three `XMLHttpRequest` and four `storage` vulnerabilities, which had not been found previously. Thus, we showcased on a labeled dataset that DoubleX is

[3]Storing and extracting data from an extension storage is not part of our attacker models, as we cannot assess to what extent that may cause damage

working in practice and is highly modular, which enables an analysis of additional dangerous APIs. In the following section, we highlight the fact that our data flow analysis also enables us to provide a much smaller number of falsely reported extensions than prior work.

6.3 Large-Scale Analysis of Chrome Extensions

To highlight the feasibility and precision of DoubleX regarding suspicious data flows detected, we apply it to Chrome extensions. In the following, we first outline how we collected 154,484 extensions and extracted their components. Subsequently, we describe our large-scale data flow analysis results with a focus on the vulnerable extensions we found.

6.3.1 Extension Collection and Setup

We designed DoubleX to analyze both Chromium-based and Firefox extensions. In this section, we focus solely on Chrome, while we discuss and analyze Firefox extensions in Section 6.4.2. We first report on our extension set before discussing the number of extensions we could analyze.

6.3.1.1 Collecting Extensions

To collect extensions, we leveraged the Chrome Web Store sitemap [35], which contains links to all extensions. Out of the 195,265 listed extensions, we could successfully download 174,112 of them on March 16, 2021.[4] The remaining extensions were either not available for download for an OS X user agent or only available for sale. Also, 19,628 downloaded extensions were themes, i.e., had no JavaScript component [48]. Thus, we retain 154,484 extensions for further consideration.

For each extension, we parsed its `manifest.json` [44] to extract the source code of its different components. Even though DoubleX can analyze UI pages, they are not part of our threat model, as they cannot be forcefully opened (i.e., attackers cannot be guaranteed to deliver their messages to UI pages). In the following, we consider content scripts, background pages, and WARs. Specifically, we combined all content scripts into a single JavaScript file. For background pages, we considered both the content of the HTML background page and the scripts listed in the manifest background section. As for WARs, we collected all HTML files flagged as accessible and extracted both inline and external scripts. In doing so, we chose to remove jQuery files (based on the output of *retire.js* [156] and file names such a *jquery-3.5.1.js*), to not analyze the well-known library to avoid running into timeouts. Finally, to ease the manual verification of our data flow reports, we leveraged *js-beautify* [122] to produce a human-readable and normalized version of each extracted file.

[4] This part of the results was updated after the initial thesis submission in October 2020

	#Analyzed	#Parsing errors	#Timeouts/Crashes
Extensions	154,484	3,674	9,500
- Content scripts	65,047	1,871	5,586
- Background pages	98,974	1,847	4,227
- WARs	7,668	597	1,454

Table 6.3: Analyzed Chrome extensions

6.3.1.2 Running DoubleX

To analyze the considered Chrome extensions, we ran DoubleX with pairwise combined content scripts/background pages and content scripts/WARs. This allows us to reason about the capabilities of a Web Attacker (who communicates with an extension through external messaging APIs or via the content scripts) as well as the ability of other extensions to send messages (since for background pages and WARs, we also analyze external messages). For performance reasons, we set a timeout of 40 minutes to analyze two components of an extension (in particular, we set four 10-minute timeouts for specific steps of our analysis). We discuss DoubleX throughput into more details in Section 6.4.1.

As indicated in Table 6.3, we could analyze 91.5% of our extension set *completely*. For 2.4%, Esprima reported errors (mostly related to syntax errors in the code or usage of the unsupported spread syntax), while 6.1% ran into a timeout or the resulting PDGs crashed the Python interpreter. Nevertheless, DoubleX could analyze the remaining extensions *partially*. While the parsing errors are specific to some extensions, the timeouts concern independent components. For example, even if the PDG generation of an extension's content script timed-out, we could analyze the background page independently for vulnerabilities. While DoubleX analyzed between 88.5 and 93.9% of the content scripts and background pages *completely*, Esprima encountered more parsing errors for WARs (7.8%), which also timed-out more often (19%). By checking the size of the WARs, we noticed that they are larger than the other components as numerous HTML files are exposed due to extensions allowing *all* files to be web-accessible.

6.3.2 Analyzing DoubleX Reports

Overall, and out of our 154,484 extension set, DoubleX reported 278 extensions as suspicious, which sums up to 309 suspicious data flows. In this section, we report on the exploitability of these flows. In particular, we aim at verifying if the suspicious data flows DoubleX reported exist. If they do, we analyze whether attackers from our threat model could exploit these flaws in practice. Subsequently, we illustrate our approach with two case studies of vulnerable extensions, which DoubleX detected. Subsequently, we discuss the evolution of vulnerable extensions between 2020 and 2021 and discuss vulnerability disclosure. Finally, we compare our detection accuracy to directly related work.

Dangerous API	#Reports	#DF	#1-way DF	#Exploitable
Code Execution	113	102	-	63
- eval	38	34	-	30
- setInterval	1	1	-	0
- setTimeout	18	15	-	1
- tabs.executeScript	56	52	-	32
Triggering Downloads	21	21	-	21
Cross-Origin Requests	95	75	11	49
- ajax	6	6	0	5
- fetch	4	4	0	3
- get	4	4	0	3
- post	1	1	0	1
- XMLHttpRequest.open	80	60	11	37
Data Exfiltration	80	77	-	76
- bookmarks.getTree	31	29	-	29
- cookies.getAll	23	23	-	22
- history.search	23	22	-	22
- topSites.get	3	3	-	3
Sum	309	275	11	209

Table 6.4: DoubleX findings on Chrome extensions

6.3.2.1 Suspicious Data Flows

As mentioned in Section 6.2.4.3, DoubleX produces fine-grained data flow reports. In particular, they contain precise information about extension components, line numbers, and corresponding computed values, where a potentially dangerous data flow was detected. Thus, we could directly look for the code logic at precise line numbers to verify what happens in practice with data from/to the attackers. Table 6.4 summarizes our findings. For each dangerous API (with a subtotal per flaw category), we first indicate the number of data flow reports (#Reports), which DoubleX generated. Subsequently, we present the results of our manual analysis, regarding the number of reports with a data flow between the attackers and a dangerous API (#DF), the number of reports with a data flow between the attackers and a specific API, but not back to the attackers (#1-way DF),[5] and finally the number of reports that attackers could exploit based on our threat model (#Exploitable).

For the code execution APIs, the majority of the reports (102 / 113) contains an attacker-controllable flow to a sink, and 63 can be confirmed as vulnerable. Regarding download triggering, all of our 21 reports have a verified dangerous data flow and could be exploited to download arbitrary files. For cross-origin requests, we can exploit 49 / 95 flaws, even though 75 have a confirmed dangerous data flow (both from and back to an attacker). In such cases, the attacker could only control a part of the URL (hence the

[5]Only relevant for the cross-origin request APIs such as `XMLHttpRequest`, where the attackers aim at executing arbitrary code and getting the response back from the server

data flow), while we aim at making *arbitrary* requests. We also observe 11 reports where an attacker could make any request but did not receive the response (but a status code, for example). Finally, regarding data exfiltration, we can exploit almost all dangerous data flows reported (76 / 80).

All in all, out of 309 reports, 275 (89%) have a full data flow between dangerous APIs and attackers, and 11 cross-origin requests have a data flow from the attackers to the sink but not back to the attackers. Therefore, we only have 23 reports without any data flow. In such cases, we observe a common limitation related to data flow over-approximation, e.g., we handle data flows at the object level and not at its properties. While our static analysis is neither sound nor complete, in the spirit of soundiness [125], we chose a trade-off between accuracy and run-time performance. Also, our approach is more oriented toward detecting suspicious data flows and less toward proving the absence of vulnerability. Specifically, we could detect 184 vulnerable extensions, totaling 209 vulnerable data flows, and impacting between 2.4 and 2.9 million users. Notably, almost 40% of these extensions can be exploited by *any* website or extension. Overall, 172 extensions are susceptible to a Web attacker, and 12 extensions are exploitable through an unprivileged extension. Besides, we could confirm 89% of the suspicious flows reported by DoubleX. While it is more challenging to evaluate the vulnerable extensions we may have missed, our analysis on the ground-truth dataset in Section 6.2.4.4 showed a true-positive rate of 92.64%. Overall, this highlights both our high precision and recall.

6.3.2.2 Case Studies

Based on our data flow reports and findings, we now describe two case studies regarding vulnerable extensions that DoubleX detected. In doing so, we underline the versatility of our tool in detecting non-obvious vulnerabilities.

Arbitrary Downloads with a Confused Deputy The extension *eflehphffapiajamoknfnpfapdgaefffk* registers an external message handler but does not specify the `externally_connectable` field in its manifest. Therefore, the handler accepts messages from any extension. The messages are then forwarded to several functions before ending in the *url* property of the `downloads.download` API, which allows an attacker to download arbitrary files. This example highlights the dangers of implicitly allowing any extension in externally connectable message handlers.

Arbitrary Code Execution The extension *cdighkgkcaadmonmbocgpcnenffjjdfc* can be exploited by *any* website to execute *arbitrary* code in the extension context. In fact, the content script, which can receive messages from any website, forwards all messages to the background page. In the background page, the messages subsequently flow into the *code* property of `tabs.executeScript`, without any sanitization. This example highlights the dangers of trusting input data which can be provided by an attacker.

6.3.2.3 Comparison Between 2020 and 2021

In this section, we discuss the evolution of vulnerable extensions between 2020 and 2021. Specifically, we focus on the life cycle of vulnerable extensions, i.e., whether vulnerable

extensions from 2020 are still in the store in 2021 and, if so, whether they are still vulnerable. To perform these analyses, we crawled the Chrome Web Store also on June 19, 2020. Of the 166,513 extensions we could extract, 132,231 (79%) were still present in the store on March 16, 2021 (and 65,546 had not been updated in this nine-month time frame).

Similarly to Section 6.3.2.1, we ran DoubleX on our 2020 extension set. Specifically, our tool flags 279 extensions (0.17%) as having a suspicious data flow, which is similar to our results from 2021 (278 / 154,484 extensions). These 279 suspicious extensions expand to 317 suspicious data flows. As previously, we manually reviewed all reports from 2020, and we confirm that 286 (90%) have a verified dangerous data flow between an attacker and the sensitive APIs we consider. As already highlighted for our 2021 extension set, DoubleX has a very high precision regarding flagged data flows. Besides, we could exploit 219 of these flows, which leads to 193 vulnerable extensions. While we found 184 vulnerable extensions in 2021, the overall number of extensions in the Chrome Web Store slightly decreased in 2021, so that the proportion of vulnerable extensions did not change between 2020 and 2021 (0.12% of extensions). Still, 30 extensions that were vulnerable in 2020 are not in the store anymore, and only three have been fixed (by removing permissions or the vulnerable API call; we discuss disclosure in the following section). While there are 19 new vulnerable extensions, which were not in the store in 2020, five extensions existed before but turned vulnerable in 2021 (three due to permission changes, one due to the addition of a vulnerable API call, and one due to allowing the communication with web pages directly in the background page).

Overall, we observe that 87% of the extensions that are vulnerable in 2021 were already vulnerable in 2020 (even though half of them were updated in between). Thus, we need a system like DoubleX to prevent vulnerable extensions from entering the store in the first place (we discuss integrating DoubleX in Chrome's vetting process in Section 6.4.3), especially as they tend to stay in the store. This is confirmed by the fact that the majority of developers we contacted did not take any action after our disclosure, as discussed in the next section.

6.3.2.4 Disclosure to Developers

We contacted the Chrome extension developers regarding the vulnerable extensions we detected. Due to the impact of the flaws, we focussed on the extensions that can be exploited by *any* website or extension, e.g., leading to arbitrary code execution in *any* website. We first reported our findings, including proof-of-concept exploits, regarding vulnerable extensions from 2020. On October 3, 2020, we contacted 22 developers via emails, 4 over contact forms, and reported 9 issues directly to Google when we did not have any contact information. As of June 2021, the developers of one extension acknowledged the bug and asked us to help them fix it. In addition, one extension (> 300k users) was updated to remove the `<all_urls>` permission to only allow the sites related to the extension, thus limiting the damage to third parties. Finally, another extension (> 50k users) fixed the vulnerability (arbitrary read). Similarly, we reported 13 additional vulnerable extensions in 2021. On May 4, we contacted 9 developers via emails and reported 4 issues directly to Google. As of June 2021, one developer acknowledged the vulnerabilities.

6.3.2.5 Comparative Analysis

Finally, we compare our approach to related work. The only prior work that leveraged a similar threat model and conducted a large-scale analysis on Chrome extensions is EmPoWeb. Hence, we contrast our results against EmPoWeb's here and defer discussion of additional related work to Chapter 7. For this comparison, we ran the open-source version of EmPoWeb [185] on our 154,484 extensions and filtered its reports to keep only the dangerous APIs that are part of our threat model. Similarly, we excluded our WARs reports from this analysis, as EmPoWeb does not consider them; i.e., we analyze the same APIs and the same extension set; hence our overall results differ slightly from Section 6.3.2.1).

As expected, EmPoWeb flags significantly more extensions as suspicious than DoubleX: it reports 2,665 extensions compared to our 268 (corresponding to 4,379 reported flaws vs. 299). Since EmPoWeb does not rest on a data flow analysis to generate reports, it mostly over-approximates the presence of an external message and of a sensitive API as a potential flaw. With DoubleX, though, we would flag such an extension only if we can find a data flow between attackers and critical APIs, hence yielding significantly fewer reports and significantly fewer false positives. In addition to being significantly more precise than EmPoWeb, DoubleX also detects vulnerabilities that EmPoWeb misses. Specifically, if we consider the 204 reports that we found vulnerable after manual review, 27 of them (13%) are not reported by EmPoWeb. It is especially prevalent for the `cookies.getAll` API, where 7 out of 22 flaws are not detected and `tabs.executeScript` (9 / 32).

We assume that our data flow and pointer analyses are responsible for our significantly better vulnerable extension detection rate. In fact, we recognize dangerous APIs and message-passing APIs even if they are not written in plain text, because we compute their values, while Somé rested on a fixed list of possible ways to invoke them. In addition, we handle calls by reference and aliasing (such as when developers redefine message listeners, e.g., to support multiple APIs, including the deprecated ones), while EmPoWeb misses such messages too.

6.3.3 Summary: Benefits of DoubleX

To sum up, out of the 154,484 Chrome extensions DoubleX analyzed, it reported only 278 extensions (0.18%) as having a data flow between the attackers and the dangerous APIs we considered. These suspicious flows expand to 309 reports, 275 (89%) of which have a verified dangerous data flow. Therefore, DoubleX is highly accurate to detect suspicious data flows. In addition, we verified that we could exploit 209 reports according to our threat model. These 209 flaws correspond to 184 vulnerable extensions, with a total of over 2.4 million users. Regarding vulnerable extensions we may have missed, we evaluated DoubleX on the vulnerable extension set provided by EmPoWeb, where we accurately flag almost 93% of the flaws (cf. Section 6.2.4.4).

In addition, we observed that 87% of the vulnerable extensions we detected in 2021 were already in the store and vulnerable one year ago (despite disclosure and half of the extensions being updated in between). As extension developers do not necessarily

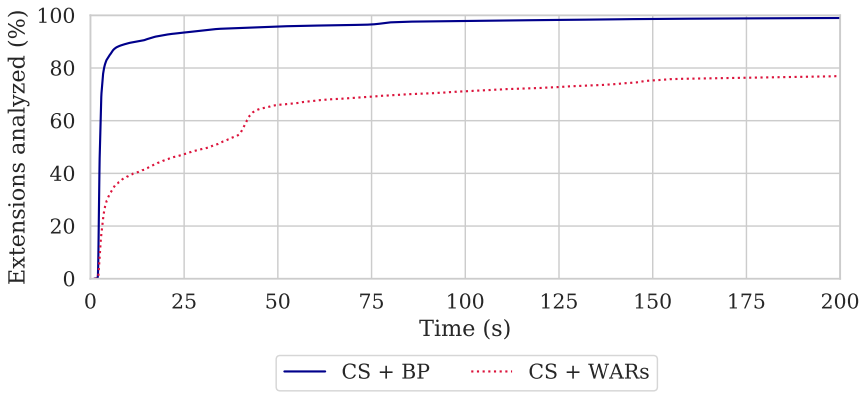


Figure 6.3: Run-time performance of DOUBLEX

have any incentive to patch vulnerable extensions, we believe that DOUBLEX could be integrated into the vetting process already conducted by Google (cf. Section 6.4.3).

6.4 Practical Applicability of DOUBLEX

We highlighted previously the precision (89%) and recall (92.64%) of our data flow analysis, and we showcased its added value compared to related work. In this section, we focus on DOUBLEX's applicability in practice. Specifically, with a median run-time of 2.5 seconds per extension, we can perform an analysis at scale. Besides Chromium-based extensions, we then showcase its applicability to other ecosystems like Firefox extensions. Finally, to identify suspicious extensions before large-scale deployment and complement current vetting workflows (focussing on malicious extensions), we envision that DOUBLEX could be integrated into Chrome vetting system.

6.4.1 Run-Time Performance

We evaluated DOUBLEX run-time performance on a server with four Intel(R) Xeon(R) Platinum 8160 CPUs (each with 48 logical cores) and a total of 1.5 TB RAM. Since DOUBLEX runs the analysis of each extension on a single core, the run-time reported is for a single CPU only.

As expected, the most time-consuming step of our approach is related to the data flow and pointer analyses. These operations naturally highly depend on the AST size as we traverse it to store the variables newly declared or look for variables previously defined, and we retraverse functions whenever they are called. On average, DOUBLEX needs 11 seconds to analyze an extension with content scripts and background pages, and 96.5 seconds for content scripts and WARs (as the WARs are larger, cf. Section 6.3.1.2). Still, the corresponding median times are 2.5 and 31.8 seconds, while the maximum amount of time are 1,498 and 1,116 seconds. In practice, our average results are heavily biased by a few extensions, whose analysis lasted a long time (but still under our 40-minute timeout). Figure 6.3 presents the Cumulative Distribution Function (CDF) [136] for our run-time performance. In particular, we could analyze 93% of our extension set for

Dangerous API	#Reports	#DF	#1-way DF	#Exploitable
ajax	1	1	0	0
downloads.download	3	3	-	3
eval	2	2	-	0
fetch	4	3	1	1
setTimeout	5	5	-	0
tabs.executeScript	2	2	-	1
XMLHttpRequest.open	7	6	1	3
Sum	24	22	2	8

Table 6.5: DOUBLEX findings on Firefox extensions

content scripts (CS) and background pages (BP) in less than 20 seconds and 45% for the content scripts and WARs. This way, DOUBLEX can effectively analyze extensions from the wild, with an analysis time of mostly a few seconds per extension.

6.4.2 Analyzing Firefox Extensions

Besides Chromium-based extensions, DOUBLEX can also analyze Firefox. In particular, we consider Firefox extensions using the WebExtension API (a cross-browser technology, compatible with the API supported by the Chromium-based browsers) [138], and not the deprecated XPCOM interface [154]. To collect these extensions, we visited the Firefox gallery [137], which contains links to all extensions, ordered per category. We used Puppeteer [171] to automatically download and unpack the extensions. We crawled the store on April 6, 2021, and could successfully collect 19,577 extensions. As for Chrome, we parsed the `manifest.json` of each extension to extract their components and ran DOUBLEX on them. Table 6.5 summarizes our findings. Out of 24 reports, we detected 8 that are exploitable under our threat model. As for Chrome, we flagged more data flows (22); for example, and as previously, we consider that merely controlling a URL prefix for an `ajax` request is not exploitable. As mentioned in Section 6.2.3.1, we took into account the specific message-passing APIs for Firefox and handled responses with a `Promise`. For the exfiltration APIs, though, we still look for callbacks and leave the `Promise` implementation for future work.

6.4.3 Extension Vetting: Workflow Integration

Given the performance of DOUBLEX, we believe that it can be integrated into the vetting process already conducted by Google for newly uploaded extensions [43]. Currently, this system aims at identifying extensions that request powerful permissions or are clearly malicious, e.g., by spreading malicious software. Still, we envision that a feedback channel to alert developers regarding potential vulnerabilities would be relevant. It is particularly important for extensions that have privileges such as `<all_urls>`, which would allow an attacker who can exploit them to make arbitrary and authenticated requests and leak their content. As Google readily points out, high-privilege extensions

require a more thorough analysis; hence, the information about vulnerabilities can also be of interest to the auditor, to limit the number of vulnerable extensions entering the store. This is all the more important as we noticed that very few developers acknowledged and fixed the vulnerabilities we reported (cf. Section 6.3.2.4).

6.5 Summary

While the three previous chapters focussed on JavaScript samples that are inherently *benign* or *malicious*; in this chapter, we highlighted the fact that attackers can also leverage *benign-but-buggy* code to perform malicious activities. In particular, we considered browser extensions, which are–due to their high privileges–a target of choice for malicious actors (*RQ4*). With our static analyzer DoubleX, we studied to what extent a malicious web page, or a malicious extension without any specific privilege, could exploit the capabilities of a vulnerable extension. Specifically, we provide a semantic abstraction of extension source code, including control and data flows, pointer analysis, and we model the intricate interactions within and outside of an extension. This way, we can perform a data flow analysis to detect suspicious flows between external actors (i.e., potential attackers) and security- and privacy-critical APIs. In practice, we analyzed 154,484 Chrome extensions and flagged 278 as having suspicious data flows. Subsequently, we verified that 89% of these flows could effectively be influenced by attackers. Overall, we detected 184 extensions that are vulnerable and lead to, e.g., arbitrary code execution in an extension privileged context. Finally, and due to DoubleX fine-grained data flow reports, we hope to raise the awareness of extension developers toward these security and privacy issues and contribute to detecting such flaws before large-scale deployment.

This chapter concludes the research work we implemented and evaluated throughout this thesis to answer our four research questions. In the following chapter, we present and discuss related work, and we underline the added value of our contributions.

7
Related Work

In this chapter, we discuss research work related to this thesis. We first present approaches to analyze and detect malicious JavaScript samples. While our primary focus is static detectors, we also highlight some dynamic systems. Next, we discuss limitations of learning-based malware detectors, which are vulnerable to adversarial attacks. Subsequently, we present different academic work that leverages the AST or PDG for vulnerability detection. Finally, we focus on the browser extension ecosystem and discuss different approaches to detect vulnerable extensions.

7.1 Malicious JavaScript Detection and Analysis

As explained in Section 2.2, static and dynamic analysis are two possible ways to detect and analyze malicious JavaScript samples.

7.1.1 Static Detectors

In the literature, several approaches have been proposed to characterize JavaScript inputs by means of static features. Such techniques mostly rely on lexical units or leverage the AST for an improved analysis. At the same time, these lexical or syntactic features can also be used to detect obfuscated JavaScript inputs. More recently, we also noted that new systems are using features with more semantic information, such as control and data flows, to detect malicious JavaScript instances.

In 2010, Rieck et al. developed Cujo, which combines n-gram features from JavaScript lexical units with dynamic code features before using an SVM classifier to detect malicious JavaScript samples [173]. As highlighted in Chapters 3 and 4, both JaSt and JStap perform better than Cujo, mainly due to lexical units lacking context information. Stock et al. also used tokens for their analysis in 2016. In particular, they implemented Kizzle, a malware signature compiler to cluster and detect exploit kits [198]. In 2011, Canali et al. worked on a faster collection of malicious web pages with Prophiler [27]. For this purpose, they leverage HTML-derived lexical features, the JavaScript AST, and an URL-based analysis to discard benign pages. With JStap's pre-filtering pipelines, we also aim at quickly and accurately classifying JavaScript inputs to send only samples with conflicting labels to more costly dynamic components. Given our differing application level (i.e., web pages vs. JavaScript), our features are naturally different. Also, Canali et al. extract very specific features from the AST, such as the number of calls to `eval` or other built-in functions, which may have become more challenging at the time of writing, due to malicious code getting more obfuscated. Specifically, to address the issue of obfuscation and dynamic code generation, in 2015, Wang et al. first deobfuscated the code to analyze before focusing on specific attacks [214]. To this end, they combine specific static features based on a textual analysis and on the AST, and dynamic features that leverage browser-level system calls.

In fact, relying on the AST is also a way to analyze JavaScript samples. In particular, Curtsinger et al. implemented Zozzle in 2011, which combines the extraction of features from the AST and their corresponding node value with a Bayesian classifier to detect malicious JavaScript [50]. To address the issue of obfuscation, Zozzle is integrated with the browser JavaScript engine to collect and process the code created at run-time.

On the contrary, JaSt and JStap do not need a deobfuscation pre-processing step and can accurately analyze both benign and malicious obfuscated files. Also, due to our n-gram analysis, we leverage the overall code syntactic structure instead of handling syntactic units separately, which leads to higher detection performance. A naive Bayes classification algorithm was also used by Hao et al. in 2014 to analyze JavaScript code by benefitting from extended API symbol features by means of the AST [75].

Beyond leveraging lexical units or the AST to detect malicious JavaScript inputs, additional work focuses on code obfuscation. For example, in 2009, Likarish et al. defined specific lexical features, based on detecting obfuscation, to recognize malicious JavaScript instances [123]. Nevertheless, in 2011, Kaplan et al. quantified the fact that obfuscation does not imply maliciousness [101]. For this purpose, they introduced NoFus, their bayesian classifier trained over the AST (similarly to Zozzle, based on syntactic units combined with their corresponding node values) to distinguish obfuscated from non-obfuscated JavaScript code. Similarly, with JSOD, Blanc et al. proposed an anomaly-based detection system over the AST to detect obfuscated scripts including readable patterns (2012) [19].

More recently (and after the publication of JStap), several systems have been proposed to detect malicious JavaScript inputs by means of a static analysis containing semantic information. Specifically, Liang et al. proposed JSAC in 2019, which combines program analysis techniques with a deep learning approach [121]. To capture both syntactic and semantic features, they combine the AST with a tree-based convolutional neural network (CNN) and the CFG with another graph-based CNN. Similarly, in 2020, Song et al. analyzed previously deobfuscated JavaScript samples with a deep learning approach on features extracted from the PDG, for preselected critical APIs [187]. The main advantage of our detectors JaSt and JStap is related to their capability of handling obfuscated inputs, which avoids this costly, mostly manual, deobfuscation step.

7.1.2 Dynamic Detectors

Additional analyses instead rely on dynamic approaches to process (malicious) JavaScript code. Given their usage of dynamic features, they are not the main focus of this thesis. Still, they are relevant for, e.g., JStap, which could pre-filter JavaScript samples to send only those with conflicting labels to more costly dynamic components, and for HideNoSeek, as the misclassified benign samples used for our camouflage attack could be analyzed by such dynamic systems. In 2005, Hallaraker et al. monitored JavaScript code execution and compared it to high-level policies to detect malicious behavior [73]. Similarly, with JSAND, Cova et al. combined anomaly detection with emulation to identify malicious JavaScript instances by comparing their behaviors to benign established profiles (2010) [49]. In 2011, Heiderich et al. proposed IceShield to execute JavaScript code in the context of a browser before leveraging machine learning techniques to detect specific attacks [76]. With EarlyBird, Schütt et al. presented their dynamic approach, combined with machine learning, to optimize the time to detect malicious JavaScript in order to limit the damage malicious code execution could already have done (2012) [176]. To improve the code coverage of dynamic approaches and detect environment-specific malware, Kolbitsch et al. implemented Rozzle in

2012 [108]. This tool imitates multiple browser and environment configurations to explore various execution paths to detect malicious JavaScript dynamically. Similarly, J-Force from Kim et al. (2017) [105] and JSForce from Hu et al. (2018) [85] also force the JavaScript execution engine to test all execution paths systematically. Beyond pure malicious JavaScript detection, Invernizzi et al. introduced EvilSeed, in 2012, to search the Web for pages likely to be malicious, by similarity detection and relation to an initial set of malicious seeds [87]. Regarding similarity detection, Kapravelos et al. presented *Revolver* in 2013, which aims at detecting evasive malicious JavaScript instances. To this end, they leverage the ASTs of the considered files to identify similarities between them while *Revolver* dynamic detector labels the files [103]. If similar files are labeled differently, they are reported as evasive. As discussed in Section 5.4.4.2, the system is not available anymore; thus, we could not evaluate HideNoSeek on it.

7.2 Adversarial Attacks

We discussed previously several ways to detect malicious JavaScript samples. In this section, we present different approaches, which aim at evading learning-based malware detectors. In particular, the difficulty of generating adversarial examples in the malware field revolves around the feature mapping function, which is traditionally not invertible. Therefore, contrary to, e.g., the image domain, it is not clear which impact some transformations in the feature space may have in the malware space. Specifically, the generated adversarial samples should still be valid code, able to run, and with the expected malicious behavior.

Previous attacks against learning-based malware detectors mostly fit into two categories. White-box attacks consider very strong attackers, which have various information about the system they are trying to evade, e.g., knowledge about the training dataset or the target model internals. For example, in 2005, Lowd et al. discussed the task of learning enough information about a given classifier to construct adversarial examples tailored for it before evaluating their approach against spam filtering [127]. On the contrary, with black-box attacks, malicious actors do not have any insider information but have a specific target system and access to the classification scores assigned to input samples or at least to the classifier's predictions. By design, both attacker models need a specific victim classifier as well as information about the system. With HideNoSeek, we go one step further and propose a generic and novel attack against static malicious JavaScript detectors. While it evades purely AST-based pipelines by construction, we also showcased its effectiveness against more complex features, such as control and data flow, as well as combinations of classifiers. In fact, and contrary to previous approaches, HideNoSeek does not try to statistically enhance the proportion of benign features in a malicious file (or put a malicious sample in a significantly bigger benign one) but exactly reproduces existing benign JavaScript ASTs, which, by construction, foils most static detectors.

In the following, we present several approaches that have been proposed to evade targeted learning-based malware detectors. These attacks all need to have at least a black-box access to the system they are trying to evade. We focus on attacks targeting PDF and Android malware detectors.

In 2012, Smutz et al. discussed an attack scenario where the attackers know the most informative features of a PDF malware classifier so that they can change the features to mimic a normal distribution [183]. Nevertheless, they performed their attack in the feature space and did not generate any evasive variants in the malware space. Instead of reproducing benign features, Maiorca et al. injected malicious content in benign PDF documents to introduce minimum differences within their benign structures while retaining the malicious behavior (reverse mimicry, 2013) [129]. On the contrary, in 2013 and 2014, Šrndić et al. modified malicious PDF documents to mimic the features of a chosen benign target [190]. In addition, they explored the strategy of training a substitute model to find evasive inputs. They also assessed the security of learning-based detection techniques by studying the range of possible attacks, according to the information leaks an attacker might have [189]. In 2016 and 2017, Xu et al. [228] and Dang et al. [51], respectively, developed a system, which stochastically manipulates malicious samples to find a variant, preserving the malicious behavior (i.e., they needed an oracle) while being classified as benign by the targeted PDF malware detectors (i.e., they also needed a black-box access to the targeted detector). Besides pure adversarial attacks, Chen et al. presented an approach to increase the cost of evading PDF malware classifiers (2020) [34]. To this end, they evaluated their system against different attacker settings.

In addition, several attacks have been proposed against the Android static malware detection system Drebin [6]. Specifically, in 2017, Grosse et al. adapted the algorithm of Papernot et al. [160] (2016)–initially defined for images–to determine which features should be changed to craft adversarial samples in the malware field [72]. In 2019, Demontis et al. focused on the transferability of evasion and poisoning attacks under different threat models. They tested their approach on Drebin as well as images, and a face recognition system [53]. They also built a Secure SVM (Sec-SVM) learning algorithm to impede evasion approaches [52]. In particular, Pierazzi et al. proposed a formalization of evasion attacks in different problem spaces, including the malware space, before testing their approach on Drebin and Sec-SVM (2020) [166].

7.3 Data Flow Analysis for Vulnerability Detection

In this thesis, we used different static code representation techniques, e.g., AST or PDG, to abstract the source code of malicious and benign JavaScript files, as well as browser extensions, before analyzing these JavaScript instances. Such techniques are also used in the field of security analysis, such as vulnerability detection. In particular, Jovanovic et al. focussed on PHP in 2006 and implemented Pixy, a static data flow analysis tool, to discover cross-site scripting vulnerabilities [92]. In 2012, Holler et al. proposed LangFuzz, their approach to automatically discover vulnerabilities, which they tested against a JavaScript and a PHP interpreter [81]. While the fuzzing ecosystem is out of the scope of this thesis, their analysis is related to HideNoSeek. In fact, they aimed at automatically generating JavaScript samples based on inputs known to have caused invalid behavior before. To this end, they replaced a given code fragment of an input file with a fragment of the same type (according to the grammar) known to have led to vulnerabilities. With HideNoSeek, we replace a benign sub-AST by a

syntactically equivalent malicious one (with respect to control and data flows) before adjusting the remaining code to generate adversarial examples. In 2012, Yamaguchi et al. extrapolated known vulnerabilities using structural patterns from the AST to find similar flaws in other projects [230]. To mine a more significant amount of source code for vulnerabilities, they introduced, in 2014, the code property graph, merging AST, CFG, and PDG into a joint data structure [229]. This way, they leveraged graph traversals to model templates for known vulnerabilities in order to find similar flaws in other projects. This new data structure was also used by Backes et al. to identify different types of web application vulnerabilities (2017) [10]. Similarly, with VulSniper, Duan et al. leveraged control flow information to encode a program as a feature tensor and feed it to a neural network to detect vulnerabilities (2019) [55]. Contrary to these approaches, to find vulnerabilities in browser extensions, DoubleX does not need any prior information.

7.4 Browser Extension Analysis

In contrast to our approach, which targets benign-but-buggy extensions, prior work rather focuses on inherently malicious extensions. Specifically, several approaches have been proposed to detect malicious extensions, e.g., by monitoring their behavior [102, 213], detecting anomalous ratings [159], or tracking the reputation of developers [88]. Such malicious behavior includes stealing users' credentials, tracking users [218], spying on them [1], and voluntarily exfiltrating sensitive user information [32].

In this thesis, as we consider benign-but-buggy extensions, they are the primary focus of this section. In the following, we discuss different approaches related to analyzing vulnerable extensions that may involuntarily lead to security and privacy issues. In 2010, Bandhakavi et al. introduced Vex, which leverages static information flow tracking on 2,452 XPCOM [154] (now deprecated) Firefox extensions [12]. Due to this Firefox infrastructure, they did not have to consider any message-passing APIs, while it is the core of our approach. Regarding Chrome, Carlini et al. combined, in 2012, a network traffic analysis of 100 extensions with a manual review to evaluate the effectiveness of Chrome security mechanisms [29]. In 2015, Calzavara et al. proposed a purely formal security analysis of browser extensions [26]. In particular, they discussed the privileges attackers might escalate if a specific extension component was compromised. In 2017, Starov et al. showcased that most privacy leakages are not intentional [193]. For this purpose, they implemented BrowsingFog and performed a dynamic analysis to detect privacy leakage from 10,000 Chrome extensions. Similarly to DoubleX, Salih et al. considered Confused Deputy-style attacks against users (2016) [25]. To this end, they leveraged the fact that XPCOM Firefox extensions shared the same JavaScript namespace, i.e., every installed extension could access the JavaScript variables defined in the global scope of all extensions. In particular, they implemented CrossFire, which performs a static data flow analysis to identify flows between globally accessible variables from extensions and security-sensitive XPCOM calls. Nevertheless, they leveraged a limitation of the XPCOM architecture (now deprecated) and, contrary to DoubleX, they did not consider the message-passing APIs as a way to exploit browser extensions' capabilities. In contrast, Somé focussed on messaging APIs to

detect vulnerable extensions (2019) [186]. For this purpose, he relied on the invocation of message handlers, combined with a lightweight call graph, to determine if they could contain dangerous API calls. Still, due to the absence of data flow tracking, he reported 3,300 suspicious extensions, only 5% of which were vulnerable in practice. With DoubleX, we go several steps further. The most noticeable difference comes from our data flow analysis, which includes control and data flow tracking, variable scoping, and pointer analysis (therefore aliasing) which leads to a much higher success rate.

7.5 Summary

In this chapter, we discussed related work and underlined the added value of our contributions. First, we presented different approaches to detect malicious JavaScript samples. Contrary to the majority of them, JaSt and JStap can handle obfuscated samples statically (i.e., they are purely static systems and do not need a dynamic or manual deobfuscation pre-processing step). Subsequently, we discussed different adversarial attacks and settings against learning-based malware detectors. Contrary to these systems, HideNoSeek is a generic attack (due to the perfect mapping onto existing benign ASTs), which does not need any prior knowledge about the systems it evades. Similarly, in contrast to previous approaches to detect vulnerabilities by means of ASTs or PDGs, DoubleX does not need any information regarding vulnerability patterns. Finally, we presented prior work to detect vulnerable Chrome and Firefox extensions. The added value of our contribution revolves around the modeling of the intricate relations between extension components, pointer analysis, and suspicious data flow tracking. This way, we limit the manual effort of verifying the reported extensions and propose a system that can statically analyze browser extensions at scale.

In the following chapter, we consider limitations of our approaches and, based on these, discuss potential future work. We finally sum up our contributions, which we link to our research questions, before concluding this thesis.

8
Conclusion

In this chapter, we first present future research work that could be performed to supplement our current approaches and results. Subsequently, we summarize our contributions and answer our initial research questions before concluding this thesis.

8.1 Discussion and Future Work

In this section, we discuss additional research directions to extend the work presented in this thesis and tackle some limitations our approaches may have. We first introduce different ideas to impede the evasion of our detectors JaSt and JStap, and to improve the detection of evasive samples (including HideNoSeek). Subsequently, we discuss adding a forced execution pipeline to analyze specific JavaScript constructs to limit over-approximations due to static analysis, without downgrading the run-time performance with too much code execution. Finally, we envision that DoubleX could be extended with additional sources and sinks to model further classes of attackers and interactions within an extension.

Detecting Evasive Samples Like all learning-based approaches, our detectors JaSt and JStap could be tampered with adversarial examples, e.g., which would mimic features of the other class or which could have been stochastically manipulated until their classification changed. To further complicate the evasion of our systems, we could leverage additional features, which would be more tamper-resistant. For example, such features could be representative of the maliciousness of a JavaScript sample at a given point in the source code. In fact, instead of analyzing the complete AST of a JavaScript input, we could split the AST to analyze, e.g., each basic block separately before computing a maliciousness score based on the maliciousness of each sub-AST.

Another class of attacks is related to the perfect mapping onto existing benign features, e.g., HideNoSeek. We showcased in Chapter 5 that static detectors are highly impacted by our attack. While combining them with dynamic systems could be a mitigation strategy, it would not scale to the large volume of JavaScript files in the wild. A static possibility might be to cluster JavaScript inputs based on their syntactic structure and to look for textual or lexical differences between instances of the same cluster. These differences could then be classified to detect the suspicious ones. Another research direction may be related to plagiarism detection. In fact, HideNoSeek reproduces benign ASTs, but there are still some, e.g., textual or data flow-based, differences between the benign and crafted files. Thus, we believe that systems measuring the similarity between input data [33] or their PDGs [124] could contribute to distinguishing benign from crafted samples. For scalability reasons, such plagiarism detection systems also work without any prior knowledge regarding the original documents that may have been partially reproduced [59].

Toward Additional JavaScript Analyses In this thesis, we performed several static analyses of JavaScript files, i.e., to detect malicious inputs and suspicious data flows in inherently benign samples. On the one hand, a static approach is fast, accurate, not environment- nor time-dependent, and provides complete coverage of the available code. On the other hand, static analysis is subject to the traditional flaws induced by

the dynamic character of JavaScript [4, 63, 89, 90, 196]. For example, JavaScript can generate code at run-time, e.g., with the `eval` function, a dynamically constructed string can be interpreted as a program fragment and executed in the current scope. In addition, JavaScript uses prototype chaining [142] to model inheritance, where properties can be added or removed during execution, and property names may be dynamically computed. While our static analyses are neither sound nor complete, in the spirit of soundiness [125], we chose a trade-off between accuracy and run-time performance.

To further improve the precision of our systems, they could be combined with additional analysis techniques, such as forced execution. To retain the scalability of our systems, specific JavaScript constructs, e.g., known to lead to over-approximations, could be selected for this dynamic analysis. The resulting execution traces could be stored in a new data structure that would enhance our current PDG with additional (run-time-based) information. In practice, we envision that having such a hybrid system could further improve the accuracy of DoubleX.

Further Securing the Extension Ecosystem DoubleX detects suspicious data flows between external actors and security- and privacy-critical APIs in browser extensions. Similarly to Mystique [32], which employs dynamic taint-tracking to detect intentional information leakage, DoubleX could also be extended to detect malicious extensions that exfiltrate data from the browser. To this end, we could retain our current implementation and update our list of sensitive sources. Also, we could consider additional ways for extension components to communicate, such as their shared extension storage. Similarly, the `localStorage` property of web pages could be considered as an extra way for an extension to communicate with external actors, which we assume would lead to the detection of additional vulnerable extensions.

8.2 Summary of Contributions

The Web has become a widespread ecosystem, interconnecting billions of people every day. Due to its popularity, it naturally also attracts the interest of attackers. Specifically, they leverage JavaScript, one of the core technologies of the Web platform, to perform malicious activities, such as drive-by downloads. While JavaScript can be abused to exploit bugs in the browser or further vulnerabilities, it is, at the same time, the most popular client-side scripting language, which is used by almost all websites to improve their interactivity and user-friendliness. Due to the widespread character of JavaScript in the wild, we need accurate systems to distinguish benign from malicious JavaScript instances at scale. Also, such detectors should neither be foiled by obfuscated inputs nor by malicious samples that are time- or environment-dependent. These reasons motivated our choice to perform a purely static analysis.

To sum up, we first showcased that we could combine features extracted from the AST with machine learning algorithms to accurately detect malicious JavaScript samples. Second, we statically generated the CFG and PDG of JavaScript programs, thus enhanced the AST with more semantic information. Combining different static code abstractions then led to a higher malicious JavaScript detection performance. Third, we showed that such static systems are vulnerable to a generic camouflage attack, which

consists in rewriting malicious ASTs into existing benign ones. Fourth, we considered another adversarial setting, where attackers leverage vulnerable browser extensions to perform malicious activities, such as executing arbitrary code in an extension's privileged context. In the following, we summarize our contributions, highlight our main findings, and answer the corresponding research questions.

8.2.1 Detecting Malicious JavaScript Through AST Analysis (*RQ1*)

To answer *RQ1: To what extent can we detect malicious (obfuscated) JavaScript inputs by combining an analysis at the AST level with machine learning algorithms?*, we developed our static AST-based analyzer JaSt. After abstracting the source code of JavaScript instances to their ASTs, we traversed the tree to extract syntactic units. Subsequently, we built 4-grams, which enabled us to preserve the units' context, and fed them to our random forest classifier.

We evaluated JaSt on a 105,305 sample set and highlighted both its high true-positive rate of 99.46% and true-negative rate of 99.48%. Due to our optimal trade-off between true positives and true negatives, we outperform the related work Cujo and Zozzle. In fact, given our usage of 4-gram features from the AST, we leverage the overall syntactic structure of the considered JavaScript files to classify them, which performs better than previous work, lacking context information. In addition, using syntactic patterns enabled us to directly handle obfuscated files. Finally, we also underlined syntactic similarities between different malicious JavaScript categories, which confirms the strength of the AST code abstraction for malicious JavaScript detection. For practical detection of malicious JavaScript samples, we make JaSt publicly available [T1].

8.2.2 Improving the Detection with Semantics in the AST (*RQ2*)

At the same time, we wondered if we could further improve our detection accuracy by going beyond relying on the sole code structure. To this end, we considered *RQ2: Can we add more semantic information into the AST of JavaScript files? Specifically, to what extent can we statically enhance the AST with control and data flows? Which features, combined with machine learning algorithms, work best to detect malicious JavaScript instances?* To address these research questions, we built JStap, our modular static malicious JavaScript detection system. Our detector combines five ways of abstracting the code, namely tokens, AST, CFG, DFG, and PDG, as well as two ways of extracting features, i.e., n-grams and node values. This way, JStap is composed of ten modules–with differing levels of context and semantic information–each of which we combined with a random forest classifier.

We evaluated our approach on a 273,216 sample set. We subsequently underlined both the high accuracy of our modules and our better detection performance compared to related work, which we reimplemented and tested on our dataset. Finally, we combined our modules to leverage the strength and knowledge of multiple code abstractions. A first pre-filtering pipeline, regrouping the predictions of a lexical, syntactic, control flow, and data flow analysis, led to unanimous predictions on almost 93% of our dataset, with a detection accuracy of 99.73%. A second pre-filtering layer on the remaining samples enabled us to classify an extra 6.5% of our dataset with an accuracy of 99%. This way,

we showcased that combining multiple code abstraction techniques further improved our accuracy. For direct deployability of our modules to detect malicious JavaScript inputs, we make JStap publicly available [T2].

8.2.3 Camouflaging Malicious JavaScript in Benign ASTs (*RQ3*)

While learning-based static systems are fast, accurate, and have a high code coverage, they are highly dependent on the static code abstraction techniques leveraged by the classifiers. This statement led to *RQ3: Can we present a generic attack against static malicious JavaScript detectors? More specifically, to what extent and how could attackers rewrite the ASTs of malicious JavaScript samples to reproduce existing benign ASTs while keeping the original malicious semantics? How effective would this camouflage be against static detectors?* To answer these research questions, we designed HideNoSeek, our generic camouflage attack. Our system first looks for syntactic clones between a benign and a malicious JavaScript input, with respect to control and data flows. Then, it replaces the reported benign sub-ASTs with their malicious syntactic equivalents. Subsequently, HideNoSeek follows the original benign data flow edges to adjust the nodes impacted by our replacement process before generating the modified AST's code. This way, HideNoSeek leverages a benign and a malicious JavaScript input to rewrite the malicious one so that its syntactic structure exactly corresponds to the benign structure while keeping the original malicious semantics.

In practice, we could generate 91,020 malicious samples, which have the same AST as Alexa top 10k websites. We then classified those crafted samples with JaSt, JStap as well as our reimplementations of Cujo and Zozzle. When the classifiers have been trained with JavaScript files from the wild (i.e., the detectors are not aware of our attack), they have false-negative rates between 99.95% and 100% on our evasive samples. On the contrary, when their training set included such crafted files, the classifiers correctly label most of our adversarial samples. In turn, they are unable to classify the benign inputs used for the camouflage as showcased by false-positive rates between 88.74% and 100%. We observed such results both for the traditional lexical and AST-based pipelines, for the control and data flow-based detectors as well as for a combination of these approaches, thus paving the way for additional research in the malware detection field.

8.2.4 Statically Analyzing Browser Extensions at Scale (*RQ4*)

Finally, beyond generating and spreading malicious JavaScript instances, attackers can also perform more stealthy, yet powerful, attacks by leveraging vulnerable browser extensions. This assumption led to *RQ4: To what extent and how can we statically analyze browser extensions to detect suspicious data flows from and toward security- and privacy-critical APIs?* To address our last research question, we developed DoubleX, which statically abstracts an extension source code to its PDG and models the interactions within and outside of an extension. More specifically, DoubleX builds the PDG of each extension component and enhances the graph with pointer analysis information to also handle aliased or slightly obfuscated API calls. We then combine these per-component PDGs with message flows between the individual components to define

a PDG at extension level. Based on our attacker models through web pages or other extensions, DoubleX then collects and leverages external messages to perform a data flow analysis to detect suspicious flows between sensitive APIs in browser extensions and external actors.

When evaluated against a ground-truth dataset, DoubleX was capable of finding 92.64% of the known vulnerabilities. We then applied DoubleX to the Chrome Web Store, where it flagged 278 / 154,484 extensions as having suspicious data flows. We could confirm that 89% of the reported flows could be controlled by external actors. Overall, we detected 184 vulnerable extensions that attackers could exploit to, e.g., execute arbitrary code in an extension context or exfiltrate sensitive user data. To raise awareness and enable developers and extension operators to automatically detect such threats, we make DoubleX publicly available [T4].

8.3 Concluding Thoughts

We hope that our work regarding the detection of malicious JavaScript instances and our resulting attack against static classifiers will pave the way to additional research in different directions. First, we highlighted the necessity for reliable and fast malware detectors. Second, there is also a need to impede the evasion of such (learning-based) systems so that they remain trustworthy in adversarial settings. In addition, we underlined the fact that malicious actors can also stealthily exploit vulnerable browser extensions. We believe that our research work could contribute to limiting such attacks. To this end, we hope to increase the awareness of well-intentioned developers toward unsafe programming practices, leading to security and privacy issues. Finally, we believe that integrating DoubleX into extension vetting systems could contribute to the detection of such flaws before the large-scale deployment of the impacted extensions, thus contribute to further protecting users' security and privacy.

Bibliography

Author's Papers for this Thesis

[P1] Fass, A., Krawczyk, R. P., Backes, M., and Stock, B. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In: *DIMVA*. 2018.

[P2] Fass, A., Backes, M., and Stock, B. JStap: A Static Pre-Filter for Malicious JavaScript Detection. In: *ACSAC*. 2019.

[P3] Fass, A., Backes, M., and Stock, B. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In: *CCS*. 2019.

[P4] Fass, A., Somé, D. F., Backes, M., and Stock, B. DoubleX: Statically Analyzing Browser Extensions at Scale. In: *Under submission*. 2021.

Author's Additional Paper

[S1] Moog, M., Demmel, M., Backes, M., and Fass, A. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In: *Dependable Systems and Networks (DSN)*. 2021.

Open-Source Implementations for this Thesis

[T1] Aurore54F. *JaSt - JS AST-Based Analysis*. `https://github.com/Aurore54F/JaSt`. Accessed on 2020-02-01.

[T2] Aurore54F. *JStap: A Static Pre-Filter for Malicious JavaScript Detection*. `https://github.com/Aurore54F/JStap`. Accessed on 2020-02-02.

[T3] Aurore54F. *HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs*. `https://github.com/Aurore54F/HideNoSeek`. Accessed on 2020-03-30.

[T4] Aurore54F. *DoubleX Research Prototype*. `https://www.dropbox.com/sh/ukxq0r4j3i33g7x/AADcD-sA73hul6VoN9w-uGdZa?dl=0`.

[T5] Aurore54F. *lexical-jsdetector*. `https://github.com/Aurore54F/lexical-jsdetector`. Accessed on 2019-11-09.

[T6] Aurore54F. *syntactic-jsdetector*. `https://github.com/Aurore54F/syntactic-jsdetector`. Accessed on 2019-11-10.

References

[1] Aggarwal, A., Viswanath, B., Zhang, L., Kumar, S., Shah, A., and Kumaraguru, P. I Spy with My Little Eye: Analysis and Detection of Spying Browser Extensions. In: *Euro S&P*. 2018.

[2] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (Second Edition)*. ISBN: 978-0321486813. Addison Wesley, 2006.

[3] Allen, F. E. Control Flow Analysis. In: *Symposium on Compiler Optimization*. 1970.

[4] Andreasen, E. and Møller, A. Determinacy in Static Analysis for jQuery. In: *Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. 2014.

[5] Andreessen, M. *Innovators of the Net: Brendan Eich and JavaScript*. `https://web.archive.org/web/20080208124612/http://wp.netscape.com/comprod/columns/techvision/innovators_be.html`. Accessed on 2020-09-28.

[6] Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., and Rieck, K. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In: *NDSS*. 2014.

[7] Arthur, D. and Vassilvitskii, S. k-means++: The Advantages of Careful Seeding. In: *ACM-SIAM Symposium on Discrete Algorithms*. 2007.

[8] AtomEditor. *Atom: a hackable text editor for the 21st Century*. `https://atom.io`. Accessed on 2019-06-05.

[9] Backes, M. and Nauman, M. LUNA: Quantifying and Leveraging Uncertainty in Android Malware Analysis through Bayesian Machine Learning. In: *Euro S&P*. 2017.

[10] Backes, M., Rieck, K., Skoruppa, M., Stock, B., and Yamaguchi, F. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In: *Euro S&P*. 2017.

[11] Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., and Vigna, G. Efficient Detection of Split Personalities in Malware. In: *NDSS*. 2010.

[12] Bandhakavi, S., Madhusudan, S. T. K. P., and Winslett, M. VEX: Vetting Browser Extensions for Security Vulnerabilities. In: *USENIX Security Symposium*. 2010.

[13] Barreno, M., Nelson, B., Joseph, A. D., and Tygar, J. D. The Security of Machine Learning. In: *Machine Learning*. 2010.

[14] Barreno, M., Nelson, B., Sears, R., Joseph, A. D., and Tygar, J. D. Can Machine Learning Be Secure? In: *AsiaCCS*. 2006.

[15] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. Clone Detection Using Abstract Syntax Trees. In: *International Conference on Software Maintenance (ICSM)*. 1998.

[16] Bergstra, J. and Bengio, Y. Random Search for Hyper-parameter Optimization. In: *Journal of Machine Learning Research*. 2012.

[17] Bhagoji, A. N., Cullina, D., Sitawarin, C., and Mittal, P. Enhancing Robustness of Machine Learning Systems via Data Transformations. In: *Annual Conference on Information Sciences and Systems (CISS)*. 2018.

[18] Biggio, B., Fumera, G., and Roli, F. Multiple Classifier Systems for Adversarial Classification Tasks. In: *International Workshop on Multiple Classifier Systems*. 2009.

[19] Blanc, G., Miyamoto, D., Akiyama, M., and Kadobayashi, Y. Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts. In: *International Conference on Advanced Information Networking and Applications Workshops*. 2012.

[20] Boutet, J. *Malicious Android Applications: Risks and Exploitation - A Spyware story about Android Application and Reverse Engineering*. Tech. rep. SANS Institute, 2010.

[21] Breiman, L. Random Forests. In: *Machine Learning*. 2001.

[22] Breiman, L., Friedman, J., Stone, C. J., and Olshen, R. *Classification and Regression Trees*. ISBN: 978-0412048418. Chapman and Hall/CRC, 1984.

[23] Brengel, M., Backes, M., and Rossow, C. Detecting Hardware-Assisted Virtualization. In: *DIMVA*. 2016.

[24] BSI. *German Federal Office for Information Security (BSI)*. `https://www.bsi.bund.de/EN`. Accessed on 2019-07-18.

[25] Buyukkayhan, A. S., Onarlioglu, K., Robertson, W., and Kirda, E. CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities. In: *NDSS*. 2016.

[26] Calzavara, S., Bugliesi, M., Crafa, S., and Steffinlongo, E. Fine-Grained Detection of Privilege Escalation Attacks on Browser Extensions. In: *Programming Languages and Systems*. 2015.

[27] Canali, D., Cova, M., Vigna, G., and Kruegel, C. Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages. In: *WWW*. 2011.

[28] CapacitorSet. *box-js - A tool for studying JavaScript malware*. `https://github.com/CapacitorSet/box-js`. Accessed on 2018-05-28.

[29] Carlini, N., Felt, A. P., and Wagner, D. An Evaluation of the Google Chrome Extension Security Architecture. In: *USENIX Security Symposium*. 2012.

[30] Carlini, N. and Wagner, D. Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods. In: *ACM Workshop on Artificial Intelligence and Security*. 2017.

[31] Carlini, N. and Wagner, D. Towards Evaluating the Robustness of Neural Networks. In: *S&P*. 2017.

[32] Chen, Q. and Kapravelos, A. Mystique: Uncovering Information Leakage from Browser Extensions. In: *CCS*. 2018.

[33] Chen, X., Francia, B., Li, M., Mckinnon, B., and Seker, A. Shared Information and Program Plagiarism Detection. In: *IEEE Transactions on Information Theory*. 2003.

[34] Chen, Y., Wang, S., She, D., and Jana, S. On Training Robust PDF Malware Classifiers. In: *USENIX Security Symposium*. 2020.

[35] chrome. *Chrome Web Store Sitemap*. `https://chrome.google.com/webstore/sitemap`. Accessed on 2021-03-16.

[36] chrome. *chrome.extension*. `https://developer.chrome.com/extensions/extension`. Accessed on 2020-05-06.

[37] chrome. *chrome.runtime*. `https://developer.chrome.com/extensions/runtime`. Accessed on 2020-02-28.

[38] chrome. *chrome.tabs*. `https://developer.chrome.com/extensions/tabs`. Accessed on 2020-05-06.

[39] chrome. *Declare Permissions*. `https://developer.chrome.com/extensions/declare_permissions`. Accessed on 2020-08-11.

[40] chrome. *Extend the Browser: Content Security Policy (CSP)*. `https://developer.chrome.com/extensions/contentSecurityPolicy`. Accessed on 2020-10-10.

[41] chrome. *Extend the Browser: Overview*. `https://developer.chrome.com/extensions/overview`. Accessed on 2020-05-09.

[42] chrome. *externally_connectable*. `https://developer.chrome.com/apps/manifest/externally_connectable`. Accessed on 2020-06-02.

[43] chrome. *How long will it take to review my item?* `https://developer.chrome.com/webstore/faq#faq-listing-108`. Accessed on 2020-10-10.

[44] chrome. *Manifest File Format*. `https://developer.chrome.com/extensions/manifest`. Accessed on 2020-06-10.

[45] chrome. *Message Passing*. `https://developer.chrome.com/extensions/messaging`. Accessed on 2020-02-28.

[46] chrome. *Migrating to Manifest V3*. `https://developer.chrome.com/extensions/migrating_to_manifest_v3`. Accessed on 2020-10-10.

[47] chrome. *The activeTab permission*. `https://developer.chrome.com/extensions/activeTab#what-activeTab-allows`. Accessed on 2020-10-10.

[48] chrome. *Themes*. `https://developer.chrome.com/extensions/themes`. Accessed on 2020-07-30.

[49] Cova, M., Kruegel, C., and Vigna, G. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In: *WWW*. 2010.

[50] Curtsinger, C., Livshits, B., Zorn, B., and Seifert, C. Zozzle: Fast and Precise In-Browser JavaScript Malware Detection. In: *USENIX Security Symposium*. 2011.

[51] Dang, H., Huang, Y., and Chang, E.-C. Evading Classifiers by Morphing in the Dark. In: *CCS*. 2017.

[52] Demontis, A., Melis, M., Biggio, B., Maiorca, D., Arp, D., Rieck, K., Corona, I., Giacinto, G., and Roli, F. Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection. In: *IEEE Transactions on Dependable and Secure Computing*. 2019.

[53] Demontis, A., Melis, M., Pintor, M., Jagielski, M., Biggio, B., Oprea, A., Nita-Rotaru, C., and Roli, F. Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks. In: *USENIX Security Symposium*. 2019.

[54] Dewald, A., Holz, T., and Freiling, F. C. ADSandbox: Sandboxing JavaScript to Fight Malicious Websites. In: *ACM Symposium on Applied Computing (SAC)*. 2010.

[55] Duan, X., Wu, J., Ji, S., Rui, Z., Luo, T., Yang, M., and Wu, Y. VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities. In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2019.

[56] Ecma International. *ECMAScript 2020 Language Specification (ECMA-262, 11th edition, June 2020)*. https://www.ecma-international.org/ecma-262/11.0. Accessed on 2020-09-28.

[57] Edwards, D. *dean.edwards.name/packer/*. http://dean.edwards.name/packer. Accessed on 2020-06-15.

[58] Egele, M., Wurzinger, P., Kruegel, C., and Kirda, E. Defending Browsers against Drive-by Downloads: Mitigating Heap-Spraying Code Injection Attacks. In: *DIMVA*. 2009.

[59] Eissen, S. M. zu and Stein, B. Intrinsic Plagiarism Analysis. In: *Advances in Information Retrieval*. 2006.

[60] Extension Monitor. *Breaking Down the Chrome Web Store*. https://extensionmonitor.com/blog/breaking-down-the-chrome-web-store-part-1. Accessed on 2020-10-07.

[61] Fawcett, T. An Introduction to ROC Analysis. In: *Pattern Recognition Letters*. 2006.

[62] Fawzi, A., Fawzi, O., and Frossard, P. Analysis of Classifiers' Robustness to Adversarial Perturbations. In: *Machine Learning*. 2015.

[63] Feldthaus, A. and Møller, A. Semi-Automatic Rename Refactoring for JavaScript. In: *Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. 2013.

[64] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1987).

[65] Fogla, P., Sharif, M., Perdisci, R., Kolesnikov, O., and Lee, W. Polymorphic Blending Attacks. In: *USENIX Security Symposium*. 2006.

[66] F-Secure. *Crypto-ransomware*. `https://www.f-secure.com/v-descs/articles/crypto-ransomware.shtml`. Accessed on 2020-09-16.

[67] Gastwirth, J. L. The Estimation of the Lorenz Curve and Gini Index. In: *Review of Economics and Statistics*. 1972.

[68] GeeksOnSecurity. *Malicious Javascript Dataset*. `https://github.com/geeksonsecurity/js-malicious-dataset`. Accessed on 2019-04-22.

[69] Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and Harnessing Adversarial Examples. In: *International Conference on Learning Representations*. 2015.

[70] Google Developers. *Closure Compiler*. `https://developers.google.com/closure/compiler`. Accessed on 2020-09-15.

[71] Grosse, K., Manoharan, P., Papernot, N., Backes, M., and McDaniel, P. On the (Statistical) Detection of Adversarial Examples. *arXiv preprint arXiv:1702.06280v2* (2017).

[72] Grosse, K., Papernot, N., Manoharan, P., Backes, M., and McDaniel, P. Adversarial Examples for Malware Detection. In: *ESORICS*. 2017.

[73] Hallaraker, O. and Vigna, G. Detecting Malicious JavaScript Code in Mozilla. In: *Conference on Engineering of Complex Computer Systems*. 2005.

[74] Han, H., Oh, D., and Cha, S. K. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In: *NDSS*. 2019.

[75] Hao, Y., Liang, H., Zhang, D., Zhao, Q., and Cui, B. JavaScript Malicious Codes Analysis Based on Naive Bayes Classification. In: *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. 2014.

[76] Heiderich, M., Frosch, T., and Holz, T. IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM. In: *RAID*. 2011.

[77] Hernández, L. *What is TrickBot?* `https://blog.f-secure.com/what-is-trickbot`. Accessed on 2020-09-17.

[78] Hickson, I. *HTML5 Web Messaging - W3C Recommendation 19 May 2015*. `https://www.w3.org/TR/webmessaging`. Accessed on 2020-09-28.

[79] Hidayat, A. *ECMAScript Parsing Infrastructure for Multipurpose Analysis*. `http://esprima.org`. Accessed on 2020-10-10.

[80] Hidayat, A. *Esprima*. `https://github.com/jquery/esprima`. Accessed on 2020-10-06.

[81] Holler, C., Herzig, K., and Zeller, A. Fuzzing with Code Fragments. In: *USENIX Security Symposium*. 2012.

[82] Hong, G., Yang, Z., Yang, S., Zhang, L., Nan, Y., Zhang, Z., Yang, M., Zhang, Y., Qian, Z., and Duan, H. How You Get Shot in the Back: A Systematical Study About Cryptojacking in the Real World. In: *CCS*. 2018.

[83] Horwitz, S., Reps, T., and Binkley, D. Interprocedural Slicing Using Dependence Graphs. In: *International Conference on Programming Language Design and Implementation (PLDI)*. 1988.

[84] Howard, F. *Malware with your Mocha? Obfuscation and Anti Emulation Tricks in Malicious JavaScript*. Tech. rep. Sophos, 2010.

[85] Hu, X., Cheng, Y., Duan, Y., Henderson, A., and Yin, H. JSForce: A Forced Execution Engine for Malicious JavaScript Detection. In: *Security and Privacy in Communication Networks*. 2018.

[86] HynekPetrak. *Javascript Malware Collection*. `https://github.com/HynekPetrak/javascript-malware-collection`. Accessed on 2019-04-22.

[87] Invernizzi, L., Benvenuti, S., Cova, M., Comparetti, P. M., Kruegel, C., and Vigna, G. EvilSeed: A Guided Approach to Finding Malicious Web Pages. In: *S&P*. 2012.

[88] Jagpal, N., Dingle, E., Gravel, J.-P., Mavrommatis, P., Provos, N., Rajab, M. A., and Thomas, K. Trends and Lessons from Three Years Fighting Malicious Extensions. In: *USENIX Security Symposium*. 2015.

[89] Jensen, S. H., Jonsson, P. A., and Møller, A. Remedying the Eval That Men Do. In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2012.

[90] Jensen, S. H., Møller, A., and Thiemann, P. Type Analysis for JavaScript. In: *International Symposium on Static Analysis (SAS)*. 2009.

[91] John, G. H. and Langley, P. Estimating Continuous Distributions in Bayesian Classifiers. In: *Conference on Uncertainty in Artificial Intelligence*. 1995.

[92] Jovanovic, N., Kruegel, C., and Kirda, E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In: *S&P*. 2006.

[93] Joyent, I. *Node.js*. `https://nodejs.org`. Accessed on 2020-09-28.

[94] Jscrambler. *Control Flow Flattening*. `https://docs.jscrambler.com/code-integrity/tutorials/control-flow-flattening?utm_source=blog.jscrambler.com&utm_medium=referral&utm_campaign=101-cff`. Accessed on 2020-06-07.

[95] jsdom. *jsdom - A JavaScript implementation of the WHATWG DOM and HTML standards, for use with node.js*. `https://github.com/jsdom/jsdom`. Accessed on 2020-06-10.

[96] Jules, D. S. *JSInspect Detect copy-pasted and structurally similar code*. `https://github.com/danielstjules/jsinspect`. Accessed on 2018-02-19.

[97] Kabilan, V. M., Morris, B., Nguyen, H.-P., and Nguyen, A. VectorDefense: Vectorization as a Defense to Adversarial Examples. In: *Soft Computing for Biomedical Applications and Related Topics*. 2021.

[98] Kachalov, T. *JavaScript Obfuscator Tool*. https://obfuscator.io. Accessed on 2020-06-04.

[99] Kafeine. *Exploit Kits*. https://malware.dontneedcoffee.com/refs/eks. Accessed on 2020-09-16.

[100] Kafeine. *MDNC - Malware don't need coffee*. https://malware.dontneedcoffee.com. Accessed on 2019-04-22.

[101] Kaplan, S., Livshits, B., Zorn, B., Siefert, C., and Curtsinger, C. *"NoFus: Automatically Detecting" + String.fromCharCode(32) + "ObFuSCateD ".toLowerCase() + "JavaScript Code"*. Tech. rep. Microsoft Research, 2011.

[102] Kapravelos, A., Grier, C., Chachra, N., Kruegel, C., Vigna, G., and Paxson, V. Hulk: Eliciting Malicious Behavior in Browser Extensions. In: *USENIX Security Symposium*. 2014.

[103] Kapravelos, A., Shoshitaishvili, Y., Cova, M., and Krügel and Giovanni Vigna, C. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In: *USENIX Security Symposium*. 2013.

[104] Kaya, J. and Rickerd, J. *Security Researchers Partner With Chrome To Take Down Browser Extension Fraud Network Affecting Millions of Users*. https://duo.com/labs/research/crxcavator-malvertising-2020. Accessed on 2020-09-17.

[105] Kim, K., Kim, I. L., Kim, C. H., Kwon, Y., Zheng, Y., Zhang, X., and Xu, D. J-Force: Forced Execution on JavaScript. In: *WWW*. 2017.

[106] KirstenS. *Cross-Site Scripting (XSS)*. https://owasp.org/www-community/attacks/xss. Accessed on 2020-09-29.

[107] Klijnsma, Y. *Inside the Magecart Breach of British Airways: How 22 Lines of Code Claimed 380,000 Victims*. https://www.riskiq.com/blog/labs/magecart-british-airways-breach. Accessed on 2018-09-14.

[108] Kolbitsch, C., Livshits, B., Zorn, B., and Seifert, C. Rozzle: De-cloaking Internet Malware. In: *S&P*. 2012.

[109] Kolter, J. Z. and Maloof, M. A. Learning to Detect and Classify Malicious Executables in the Wild. In: *Journal of Machine Learning Research*. 2006.

[110] Komondoor, R. and Horwitz, S. Using Slicing to Identify Duplication in Source Code. In: *International Symposium on Static Analysis (SAS)*. 2001.

[111] Konoth, R. K., Vineti, E., Moonsamy, V., Lindorfer, M., Kruegel, C., Bos, H., and Vigna, G. MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense. In: *CSS*. 2018.

[112] Koschke, R., Falke, R., and Frenzel, P. Clone Detection Using Abstract Syntax Suffix Trees. In: *Working Conference on Reverse Engineering*. 2006.

[113] Kotov, V. and Massacci, F. Anatomy of Exploit Kits: Preliminary Analysis of Exploit Kits as Software Artefacts. In: *International Conference on Engineering Secure Software and Systems*. 2013.

[114] Krinke, J. Identifying Similar Code with Program Dependence Graphs. In: *Working Conference on Reverse Engineering (WCRE)*. 2001.

[115] Lakshmanan, R. *49 New Google Chrome Extensions Caught Hijacking Cryptocurrency Wallets*. https://thehackernews.com/2020/04/chrome-cryptocurrency-extensions.html. Accessed on 2020-10-02.

[116] Larin, B. *Magnitude exploit kit – evolution*. https://securelist.com/magnitude-exploit-kit-evolution/97436. Accessed on 2020-09-16.

[117] Laskov, P. and Šrndić, N. Static Detection of Malicious JavaScript-Bearing PDF Documents. In: *ACSAC*. 2011.

[118] Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., and Joosen, W. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In: *NDSS*. 2019.

[119] Lecuyer, M., Atlidakis, V., Geambasu, R., Hsu, D., and Jana, S. Certified Robustness to Adversarial Examples with Differential Privacy. In: *S&P*. 2019.

[120] Lee, S., Han, H., Cha, S. K., and Son, S. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In: *USENIX Security Symposium*. 2020.

[121] Liang, H., Yang, Y., Sun, L., and Jiang, L. JSAC: A Novel Framework to Detect Malicious JavaScript via CNNs over AST and CFG. In: *International Joint Conference on Neural Networks (IJCNN)*. 2019.

[122] Lielmanis, E. *js-beautify*. https://www.npmjs.com/package/js-beautify. Accessed on 2020-07-30.

[123] Likarish, P., Jung, E., and Jo, I. Obfuscated Malicious JavaScript Detection Using Classification Techniques. In: *International Conference on Malicious and Unwanted Software (MALWARE)*. 2009.

[124] Liu, C., Chen, C., Han, J., and Yu, P. S. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In: *International Conference on Knowledge Discovery and Data Mining (KDD)*. 2006.

[125] Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B.-Y. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. In Defense of Soundiness: A Manifesto. In: *Communications of the ACM*. 2015.

[126] Logic, D. *Daft Logic: Online Javascript Obfuscator*. https://www.daftlogic.com/projects-online-javascript-obfuscator.htm. Accessed on 2020-06-09.

[127] Lowd, D. and Meek, C. Adversarial Learning. In: *International Conference on Knowledge Discovery and Data Mining (KDD)*. 2005.

[128] Lu, L., Yegneswaran, V., Porras, P., and Lee, W. BLADE: An Attack-Agnostic Approach for Preventing Drive-by Malware Infections. In: *CCS*. 2010.

[129] Maiorca, D., Corona, I., and Giacinto, G. Looking at the Bag is Not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection. In: *AsiaCCS*. 2013.

[130] McCallum, A. and Nigam, K. A Comparison of Event Models for Naive Bayes Text Classification. In: *AAAI-98 Workshop on Learning for Text Categorization*. 1998.

[131] Meng, D. and Chen, H. MagNet: A Two-Pronged Defense Against Adversarial Examples. In: *CCS*. 2017.

[132] Menn, J. *Exclusive: Massive spying on users of Google's Chrome shows new security weakness*. https://www.reuters.com/article/us-alphabet-google-chrome-exclusive/exclusive-massive-spying-on-users-of-googles\-chrome-shows-new-security-weakness-idUSKBN23P0JO?il=0. Accessed on 2020-09-17.

[133] Metzen, J. H., Genewein, T., Fischer, V., and Bischoff, B. On Detecting Adversarial Perturbations. In: *International Conference on Learning Representation (ICLR)*. 2017.

[134] Microsoft. Microsoft Security Intelligence Report V16. In: 2013.

[135] mishoo. *UglifyJS 2*. https://github.com/mishoo/UglifyJS/tree/v2.x. Accessed on 2020-09-15.

[136] Monti, K. L. Folded Empirical Distribution Function Curves (Mountain Plots). In: *The American Statistician*. 1995.

[137] Mozilla. *Firefox Browser Add-ons: Extensions*. https://addons.mozilla.org/en-US/firefox/extensions. Accessed on 2021-04-06.

[138] Mozilla Developer Network. *Browser Extensions*. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions. Accessed on 2020-06-10.

[139] Mozilla Developer Network. *Conditional (ternary) operator*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator. Accessed on 2020-06-07.

[140] Mozilla Developer Network. *EventTarget.addEventListener()*. https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener. Accessed on 2020-02-27.

[141] Mozilla Developer Network. *Functions*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions. Accessed on 2020-05-17.

[142] Mozilla Developer Network. *Inheritance and the prototype chain*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain. Accessed on 2020-10-10.

[143] Mozilla Developer Network. *JavaScript Conditional Compilation: cc_on*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Microsoft_Extensions/at-cc-on. Accessed on 2019-06-04.

[144] Mozilla Developer Network. *manifest.json: permissions*. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/permissions. Accessed on 2020-05-09.

[145] Mozilla Developer Network. *Promise*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. Accessed on 2020-10-10.

[146] Mozilla Developer Network. *Property accessors*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_accessors. Accessed on 2020-09-15.

[147] Mozilla Developer Network. *Same-origin policy*. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Accessed on 2020-05-09.

[148] Mozilla Developer Network. *SpiderMonkey*. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. Accessed on 2019-09-10.

[149] Mozilla Developer Network. *Standard built-in objects/Boolean*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean. Accessed on 2019-05-18.

[150] Mozilla Developer Network. *tabs.executeScript()*. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/tabs/executeScript. Accessed on 2020-07-30.

[151] Mozilla Developer Network. *WindowEventHandlers.onmessage*. https://developer.mozilla.org/fr/docs/Web/API/WindowEventHandlers/onmessage. Accessed on 2020-02-27.

[152] Mozilla Developer Network. *Window.localStorage*. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage. Accessed on 2020-08-11.

[153] Mozilla Developer Network. *Window.postMessage()*. https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage. Accessed on 2020-02-20.

[154] Mozilla Developer Network. *XPCOM Interfaces*. https://developer.mozilla.org/en-US/docs/Archive/Mozilla/XUL/Tutorial/XPCOM_Interfaces. Accessed on 2020-10-10.

[155] Nidecki, T. A. *Mutation XSS in Google Search*. https://www.acunetix.com/blog/web-security-zone/mutation-xss-in-google-search. Accessed on 2020-10-03.

[156] Oftedal, E. *Retire.js: What you require you must also retire*. https://retirejs.github.io/retire.js. Accessed on 2020-07-30.

[157] Palladino, P. *Non alphanumeric JavaScript*. http://patriciopalladino.com/blog/2012/08/09/non-alphanumeric-javascript.html. Accessed on 2019-06-10.

[158] Pan, X., Cao, Y., Liu, S., Zhou, Y., Chen, Y., and Zhou, T. CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites. In: *CCS*. 2016.

[159] Pantelaios, N., Nikiforakis, N., and Kapravelos, A. You've Changed: Detecting Malicious Browser Extensions through their Update Deltas. In: *CCS*. 2020.

[160] Papernot, N., McDaniel, P. D., Jha, S., Fredrikson, M., Celik, Z. B., and Swami, A. The Limitations of Deep Learning in Adversarial Settings. In: *Euro S&P*. 2016.

[161] Papernot, N., McDaniel, P., and Goodfellow, I. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *arXiv preprint arXiv:1605.07277* (2016).

[162] Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z. B., and Swami, A. Practical Black-Box Attacks Against Machine Learning. In: *AsiaCCS*. 2017.

[163] Parameshwaran, I., Budianto, E., Shinde, S., Dang, H., Sadhu, A., and Saxena, P. DexterJS: Robust Testing Platform for DOM-based XSS Vulnerabilities. In: *Foundations of Software Engineering*. 2015.

[164] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* (2011).

[165] Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., and Cavallaro, L. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In: *USENIX Security Symposium*. 2019.

[166] Pierazzi, F., Pendlebury, F., Cortellazzi, J., and Cavallaro, L. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In: *S&P*. 2020.

[167] Platt, J. C. Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods. In: *Advances in Large Margin Classifiers*. 1999.

[168] Powers, D. M. W. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. In: *Journal of Machine Learning Technologies*. 2011.

[169] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., and Modadugu, N. The Ghost In The Browser Analysis of Web-based Malware. In: *USENIX Workshop on Hot Topics in Understanding Botnets*. 2007.

[170] PublicWWW. *`toString()` Usage*. https://publicwww.com/websites/%22toString%28%29%22. Accessed on 2019-07-18.

[171] puppeteer. *puppeteer*. https://github.com/puppeteer/puppeteer. Accessed on 2020-07-30.

[172] Rieck, K. *Jassi: A Simple and Robust JavaScript Lexer*. https://github.com/rieck/jassi. Accessed on 2019-05-24.

[173] Rieck, K., Krueger, T., and Dewald, A. Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In: *ACSAC*. 2010.

[174] Rogers, S. and Girolami, M. *A First Course in Machine Learning (Second Edition)*. Chapman & Hall/CRC Machine Learning & Pattern Recognition. ISBN: 978-1498738484. Chapman and Hall/CRC, 2016.

[175] Sánchez-Rola, I., Santos, I., and Balzarotti, D. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In: *USENIX Security Symposium*. 2017.

[176] Schütt, K., Kloft, M., Bikadorov, A., and Rieck, K. Early Detection of Malicious Behavior in JavaScript Code. In: *ACM Workshop on Security and Artificial Intelligence*. 2012.

[177] scikit-learn developers. *Scikit-learn: HashingVectorizer*. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html. Accessed on 2019-06-05.

[178] scikit-learn developers. *sklearn.ensemble.RandomForestClassifier*. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.feature_importances_. Accessed on 2019-06-12.

[179] Segura, J. *Exploit kits: 2019 reviews*. https://blog.malwarebytes.com/?s=exploit%20kits%202019. Accessed on 2020-09-16.

[180] Shao, Y., Luo, X., Qian, C., Zhu, P., and Zhang, L. Towards a Scalable Resource-driven Approach for Detecting Repackaged Android Applications. In: *ACSAC*. 2014.

[181] Sjösten, A., Acker, S., and Sabelfeld, A. Discovering Browser Extensions via Web Accessible Resources. In: *Conference on Data and Application Security and Privacy (CODASPY)*. 2017.

[182] Skolka, P., Staicu, C.-A., and Pradel, M. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In: *WWW*. 2019.

[183] Smutz, C. and Stavrou, A. Malicious PDF Detection using Metadata and Structural Features. In: *ACSAC*. 2012.

[184] Smutz, C. and Stavrou, A. When a Tree Falls: Using Diversity in Ensemble Classifiers to Identify Evasion in Malware Detectors. In: *NDSS*. 2016.

[185] Somé, D. *extsanalyzer (EmPoWeb)*. https://gitlab.com/doliere/extsanalyzer. Accessed on 2020-10-10.

[186] Somé, D. F. EmPoWeb: Empowering Web Applications with Browser Extensions. In: *S&P*. 2019.

[187] Song, X., Chen, C., Cui, B., and Fu, J. Malicious JavaScript Detection Based on Bidirectional LSTM Model. In: *Applied Sciences*. 2020.

[188] Soni, P., Budianto, E., and Saxena, P. The Sicilian Defense: Signature-Based Whitelisting of Web JavaScript. In: *CCS*. 2015.

[189] Šrndic, N. and Laskov, P. Practical Evasion of a Learning-Based Classifier: A Case Study. In: *S&P*. 2014.

[190] Šrndić, N. and Laskov, P. Detection of Malicious PDF Files Based on Hierarchical Document Structure. In: *NDSS*. 2013.

[191] Staicu, C.-A., Pradel, M., and Livshits, B. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. In: *NDSS*. 2018.

[192] Starov, O., Laperdrix, P., Kapravelos, A., and Nikiforakis, N. Unnecessarily Identifiable: Quantifying the Fingerprintability of Browser Extensions due to Bloat. In: *WWW*. 2019.

[193] Starov, O. and Nikiforakis, N. Extended Tracking Powers: Measuring the Privacy Diffusion Enabled by Browser Extensions. In: *WWW*. 2017.

[194] Starov, O. and Nikiforakis, N. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In: *S&P*. 2017.

[195] StatCounter. *Desktop Browser Market Share Worldwide*. `https://gs.statcounter.com/browser-market-share/desktop/worldwide`. Accessed on 2020-10-07.

[196] Stein, B., Nielsen, B. B., Chang, B.-Y. E., and Møller, A. Static Analysis with Demand-Driven Value Refinement. In: *ACM on Programming Languages*. 2019.

[197] Stock, B., Johns, M., Steffens, M., and Backes, M. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security. In: *USENIX Security Symposium*. 2017.

[198] Stock, B., Livshits, B., and Zorn, B. Kizzle: A Signature Compiler for Detecting Exploit Kits. In: *Dependable Systems and Networks (DSN)*. 2016.

[199] Suzuki, Y. *ECMAScript Code Generator*. `https://github.com/estools/escodegen`. Accessed on 2018-06-15.

[200] Sven. *JSDetox - A Javascript malware analysis tool using static analysis / deobfuscation techniques and an execution engine featuring HTML DOM emulation*. `http://www.relentless-coding.org/projects/jsdetox`. Accessed on 2018-05-24.

[201] Symantec Security Response. *Mirai: what you need to know about the botnet behind recent major DDoS attacks*. `https://www.symantec.com/connect/blogs/mirai-what-you-need-know-about-botnet-behind-recent-major-ddos-attacks`. Accessed on 2020-09-17.

[202] Symantec Security Response. *Petya ransomware outbreak: Here is what you need to know*. `https://www.symantec.com/blogs/threat-intelligence/petya-ransomware-wiper`. Accessed on 2020-09-17.

[203] Symantec Security Response. *What you need to know about the WannaCry Ransomware*. https://www.symantec.com/blogs/threat-intelligence/wannacry-ransomware-attack. Accessed on 2020-09-17.

[204] Tate, R. *Open Source Applications and XSS Attacks*. https://www.whitehatsec.com/blog/open-source-applications-and-xss-attacks. Accessed on 2020-10-02.

[205] The MITRE Corporation. *CVE-2018-15982 | Adobe Flash Player Arbitrary Code Execution*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2018-15982. Accessed on 2020-09-16.

[206] The MITRE Corporation. *CVE-2018-4878 | Adobe Flash Player Arbitrary Code Execution*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-4878. Accessed on 2020-09-16.

[207] The MITRE Corporation. *CVE-2018-8174 | Windows VBScript Engine Remote Code Execution Vulnerability*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-8174. Accessed on 2020-09-16.

[208] The MITRE Corporation. *CVE-2019-1367 | Scripting Engine Memory Corruption Vulnerability*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1367. Accessed on 2020-09-16.

[209] The SciPy community. *scipy.sparse.csr_matrix*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html#scipy.sparse.csr_matrix. Accessed on 2019-06-05.

[210] VirusTotal. *VirusTotal - Analyze suspicious files and URLs to detect types of malware, automatically share them with the security community*. https://www.virustotal.com. Accessed on 2019-04-22.

[211] W3Techs. *Usage of JavaScript libraries for websites*. https://w3techs.com/technologies/overview/javascript_library/all. Accessed on 2018-11-13.

[212] W3Techs. *Usage statistics of JavaScript as client-side programming language on websites*. https://w3techs.com/technologies/details/cp-javascript. Accessed on 2020-10-20.

[213] Wang, J., Li, X., Liu, X., Dong, X., Wang, J., Liang, Z., and Feng, Z. An Empirical Study of Dangerous Behaviors in Firefox Extensions. In: *International Conference on Information Security (ISC)*. 2012.

[214] Wang, J., Xue, Y., Liu, Y., and Tan, T. H. JSDC: A Hybrid Approach for JavaScript Malware Detection and Classification. In: *AsiaCCS*. 2015.

[215] Wang, K., Parekh, J. J., and Stolfo, S. J. Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In: *RAID*. 2006.

[216] Wang, W., Ferrell, B., Xu, X., Hamlen, K. W., and Hao, S. SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In: *ESORICS*. 2018.

[217] Weiser, M. Program Slicing. In: *International Conference on Software Engineering (ICSE)*. 1981.

[218] Weissbacher, M., Mariconti, E., Suarez-Tangil, G., Stringhini, G., Robertson, W., and Kirda, E. Ex-Ray: Detection of History-Leaking Browser Extensions. In: *ACSAC*. 2017.

[219] West, M. *Content Security Policy Level 3 - W3C Working Draft, 15 October 2018*. `https://www.w3.org/TR/CSP3`. Accessed on 2020-09-28.

[220] WHATWG. *DOM - Living Standard - Last Updated 28 September 2020*. `https://dom.spec.whatwg.org`. Accessed on 2020-09-29.

[221] WHATWG. *HTML - Living Standard - Last Updated 28 September 2020*. `https://html.spec.whatwg.org/multipage`. Accessed on 2020-09-29.

[222] WHATWG. *HTML Origin - Living Standard - Last Updated 28 September 2020*. `https://html.spec.whatwg.org/multipage/origin.html`. Accessed on 2020-09-29.

[223] Wilson, E. B. and Hilferty, M. M. The Distribution of Chi-Squared. *National Academy of Sciences of the United States of America* (1931).

[224] Witten, I. H., Frank, E., Hall, M. A., and J.Pal, C. *Data Mining: Practical Machine Learning Tools and Techniques, fourth edition*. Data Management Systems. ISBN: 978-0128042915. Morgan Kaufmann, 2016.

[225] Wressnegger, C., Schwenk, G., Arp, D., and Rieck, K. A Close Look on n-Grams in Intrusion Detection: Anomaly Detection vs. Classification. In: *ACM Workshop on Artificial Intelligence and Security*. 2013.

[226] Xu, W., Zhang, F., and Zhu, S. The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study. In: *International Conference on Malicious and Unwanted Software (MALWARE)*. 2012.

[227] Xu, W., Evans, D., and Qi, Y. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. In: *NDSS*. 2018.

[228] Xu, W., Qi, Y., and Evans, D. Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. In: *NDSS*. 2016.

[229] Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. Modeling and Discovering Vulnerabilities with Code Property Graphs. In: *S&P*. 2014.

[230] Yamaguchi, F., Lottmann, M., and Rieck, K. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In: *ACSAC*. 2012.

[231] Yara-Rules. *Repository of Yara rules*. `https://github.com/Yara-Rules/rules`. Accessed on 2019-01-23.

[232] Youden, W. J. Index for Rating Diagnostic Tests. In: *Cancer*. 1950.

[233] Zhou, W., Zhou, Y., Jiang, X., and Ning, P. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In: *ACM Conference on Data and Application Security and Privacy*. 2012.

A
Appendix

This appendix complements Chapter 6.

In particular, Figure A.1 represents the PDG of the vulnerable content script from Listing 6.3. As explained in the corresponding Section 6.2.4.2, and as shown in the PDG, the first call to `eval` does not appear in plain text but is correctly computed by our pointer analysis module while we traverse the graph. Subsequently, DOUBLEX accurately flags this call to `eval` as having a suspicious data flow. The corresponding full report is in Listing A.1.

Finally, we illustrate an extension PDG, including control, data, and message flows in Figure A.2.

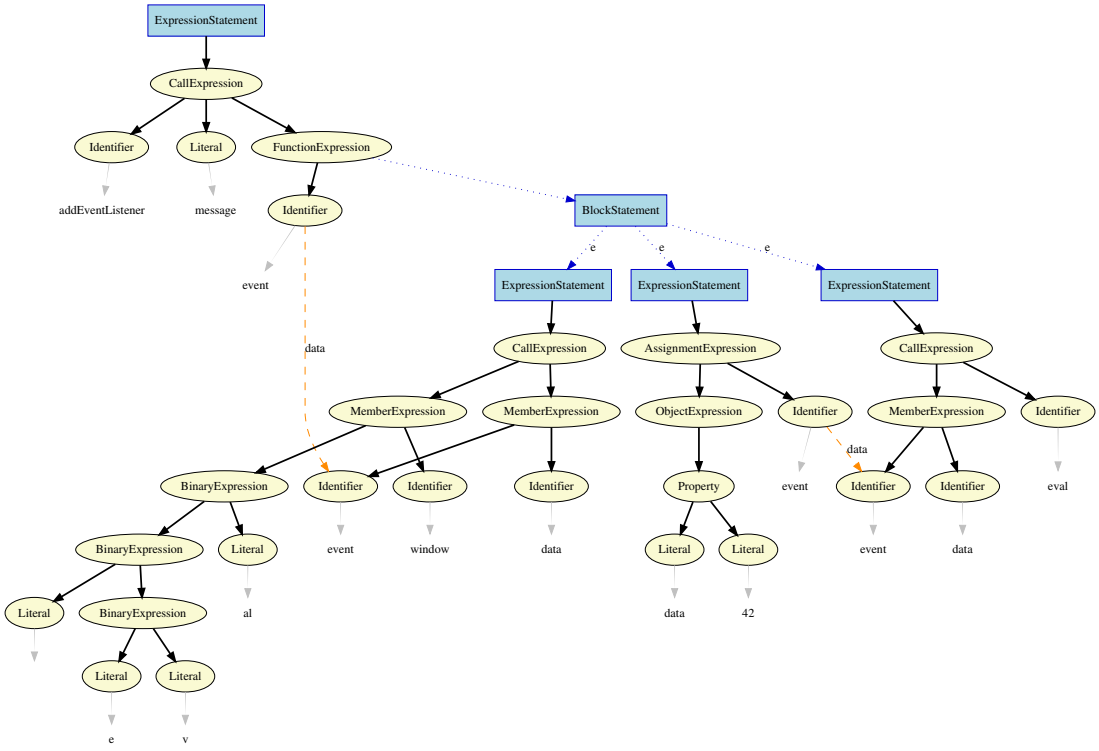


Figure A.1: AST of the vulnerable content script from Listing 6.3 extended with control & data flows

```
{
  "extension": "vuln-extension",
  "cs": {
    "direct_dangers": {
      "danger1": {
        "danger": "eval",
        "value": "window.eval(event.data)",
        "sink-param1": "event.data",
        "line": "2 - 2",
        "filename": "vuln-extension/content-script.js",
        "dataflow": true,
        "param_id0": {
          "received_from_wa_1": {
            "wa": "event",
            "line": "1 - 1",
            "filename": "vuln-extension/content-script.js",
            "where1": "event",
          }
        }
      },
      "danger2": {
        "danger": "eval",
        "value": "eval(42)",
        "sink-param1": 42,
        "line": "4 - 4",
        "filename": "vuln-extension/content-script.js",
        "dataflow": false,
        "param_id0": {}
      }
     },
    "indirect_dangers": {},
    "exfiltration_dangers": {}
  },
  "bp": {
    "direct_dangers": {},
    "indirect_dangers": {},
    "exfiltration_dangers": {}
  }
}
```

Listing A.1: Full data flow report for Listing 6.3

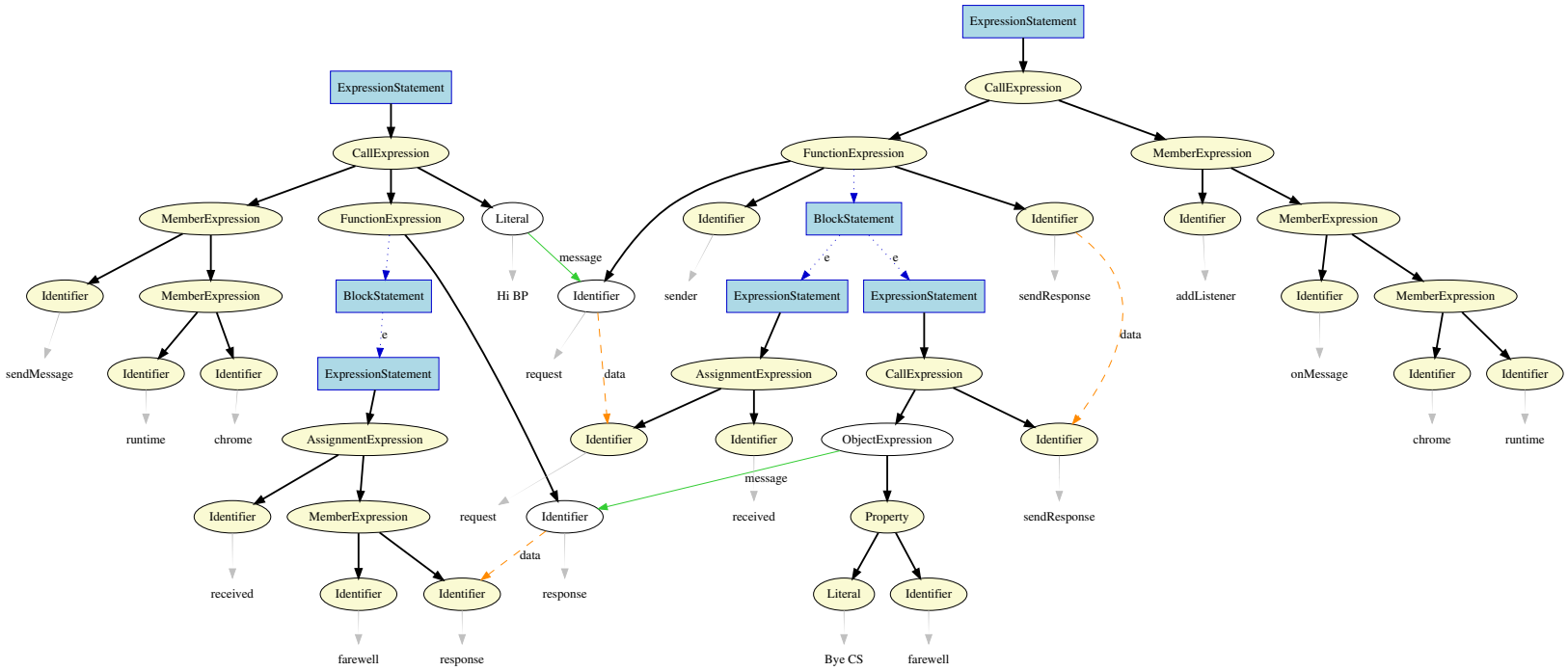


Figure A.2: AST of the extension from Listing 6.2 extended with control, data & message flows