Studying JavaScript Security Through Static Analysis

PhD Thesis Summary

Aurore Fass

This document is a summary of my PhD thesis. My full thesis is available at [8].

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.

1 Introduction

The Web has become the most popular software platform, used by billions of people every day. Given its popularity, it naturally attracts the interest of malicious actors who try to leverage the Web as a vector for attacking their victims' machines. In particular, the first step to harm a victim's machine often relies on JavaScript payloads [13, 21]. While JavaScript was initially designed to create sophisticated and interactive web pages, it is also used to perform malicious activities, such as drive-by downloads or crypto mining.

In this thesis, we focus on two orthogonal threat models: *malicious* JavaScript and *benign-but-buggy* (vulnerable) JavaScript code. We use the term *malicious* JavaScript to refer to code designed by *malicious* actors with the aim of harming victims. Ultimately malicious JavaScript will likely exploit vulnerabilities to download and execute malware. On the contrary, *benign-but-buggy* JavaScript relates to code designed by *well-intentioned* developers but which contains some vulnerabilities that malicious actors could exploit. Due to their elevated privileges compared to web pages, we chose to focus on browser extensions whose main logic is written in JavaScript. Ultimately, vulnerable JavaScript in a browser extension can lead to, e.g., universal cross-site scripting (i.e., the ability to execute code in *any* websites, even without a vulnerability in the websites themselves) or sensitive user data exfiltration to *any* websites.

Both malicious and vulnerable JavaScript code can be leveraged as attack vectors to compromise the security and privacy of Internet users. In this thesis, we develop advanced methods to uncover such attacks against end users. As dynamic analysis is costly, can be tricked by malware, and has a limited code coverage, we chose to *statically* analyze JavaScript insecurities at scale. We make four main contributions:

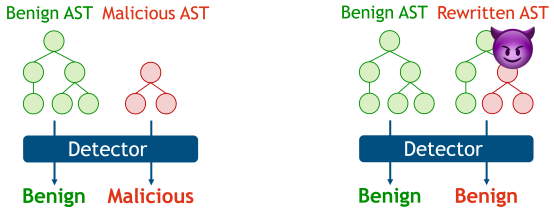
- JAST*: An AST-Based Malicious JavaScript Detector*: We present our machine learning-based system JAST, which detects malicious JavaScript with an accuracy of 99.5%.
- JSTAP*: A Static Pre-Filter for Malicious JavaScript Detection*: We introduce our modular detector JSTAP, which leverages semantic information to pre-filter JavaScript samples likely to be malicious (accuracy: 99.73%).
- HIDENOSEEK*: Camouflaging Malicious JavaScript in Benign ASTs*: With HIDENOSEEK, we propose a *generic* attack against static malicious JavaScript detectors. We show that they are inept to handle our crafted samples, which they misclassify over 99.95% of the time (false negatives).
- DOUBLEX*: Statically Analyzing Browser Extensions at Scale*: Beyond malicious JavaScript, we design DOUBLEX to detect vulnerable data flows in browser extensions. We report on 184 vulnerable extensions. In addition, we highlight DOUBLEX high precision (89%) and recall (93%).

This thesis is based on the four following papers, which have all been published at peer-reviewed conferences:

[A1] Fass, A., Krawczyk, R. P., Backes, M., and Stock, B. JAST: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In: *DIMVA*. Code repository: https://github.com/Aurore54F/JaSt. 2018.

[A2] Fass, A., Backes, M., and Stock, B. JSTAP: A Static Pre-Filter for Malicious JavaScript Detection. In: *ACSAC*. Code: https://github.com/Aurore54F/JStap. 2019.

[A3] Fass, A., Backes, M., and Stock, B. HIDENOSEEK: Camouflaging Malicious JavaScript in Benign ASTs. In: *ACM CCS*. Code repository: https://github.com/Aurore54F/HideNoSeek. 2019.



(a) Benign vs. malicious AST; those differences can be leveraged for an accurate detection

(b) Benign AST vs. malicious AST rewritten by HIDENOSEEK

Figure 1: Schematic depiction of different ASTs

[A4] Fass, A., Somé, D. F., Backes, M., and Stock, B. DOUBLEX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In: *ACM CCS*. Code repository: https://github.com/Aurore54F/DoubleX. 2021.

In the spirit of open science, for reproducibility, follow-up work, and practical detection of malicious JavaScript samples and vulnerable browser extensions, the corresponding research prototypes are all available on GitHub.

2 JAST: An AST-Based Malicious JavaScript Detector

Attackers abuse JavaScript to, e.g., exploit bugs in the browser or perform drive-by downloads. To hinder the analysis and the detection of such nefarious scripts, malicious actors take advantage of code obfuscation, which foils, e.g., signature-based detection. It has been shown that machine learning-based approaches are effective to learn features typical of benign vs. malicious JavaScript samples [5, 16] (2011). Still, the malicious JavaScript landscape has evolved since 2011, when few obfuscation schemes were being used. Besides, due to the large volume of JavaScript files in the wild, executing all of them to check for maliciousness is not realistic anymore.

To overcome these challenges, we propose JAST, our fully static system, which distinguishes malicious from benign (even obfuscated) JavaScript samples with an accuracy of almost 99.5%.

2.1 JAST

JAST abstracts the code of a JavaScript sample to its AST (Abstract Syntax Tree). This way, we eliminate the artificial noise induced, e.g., by identifier renaming, to solely focus on structural constructs. In particular, we find that benign and malicious JavaScript samples have a different AST (cf. Figure 1a). We capture such discrepancies automatically, by extracting syntactic units from the AST. To preserve the units' context, e.g., a `for` loop or a `try/catch` block, we extract substrings of length 4 (namely 4-grams), whose frequency we analyze. We find that these features differ between benign and malicious samples, enabling our random forest classifier to learn to automatically distinguish them (cf. Figure 1a).

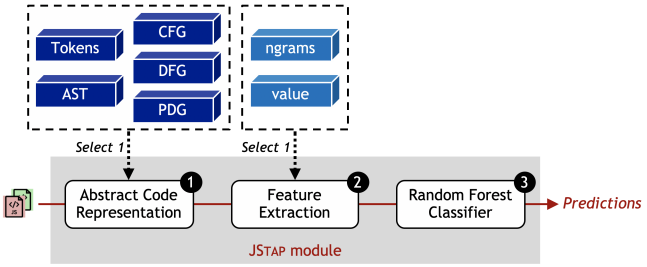


Figure 2: Architecture of JSTAP

2.2 Detecting Malicious JavaScript Samples

We evaluated JAST on over 105,000 JavaScript samples (20k benign and 85k malicious). JAST correctly classifies 99.48% of our benign dataset while still detecting 99.46% of our malicious samples, thereby outperforming related work [5, 16]. As both our benign and malicious samples are, for the most part, obfuscated, this demonstrates the resilience of our system to this specific form of evasion. In addition, we studied and discussed the evolution of JAST accuracy over one year. Finally, we leveraged different classes of malicious JavaScript samples, e.g., emails vs. exploit kits, and showcased that syntax-based features are core in classifying JavaScript inputs. Overall, these findings highlight the practical applicability of JAST to detect malicious JavaScript samples in the wild.

3 JSTAP: A Static Pre-Filter for Malicious JavaScript Detection

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. With JSTAP, we explore to what extent we can go beyond the code structure by also considering control and data flow information to detect malicious JavaScript inputs.

3.1 JSTAP

JSTAP is our modular malicious JavaScript detector. As shown in Figure 2, JSTAP can leverage five ways of abstracting JavaScript code, with differing levels of context and semantic information. In particular, JSTAP can rely on tokens (i.e., a linear conversion of the code into abstract symbols), the AST, CFG (Control Flow Graph), DFG (Data Flow Graph), or PDG (Program Dependency Graph). We generate these graphs statically by adding control and/or data flow edges to the AST.[1] In addition, JSTAP regroups two ways of extracting features: n-grams (as for JAST) or considering variable value information. This way, JSTAP is composed of ten modules, which can be used separately or combined. For each module, we finally train a random forest classifier to distinguish benign from malicious JavaScript inputs.

[1] In our thesis, we thoroughly describe and illustrate how we build these graphs and discuss to what extent they differ from the original definition of Allen [1] and Ferrante et al. [9].

3.2 JSTAP Module Combination

We evaluated JSTAP on our dataset totaling over 270,000 JavaScript samples (130k malicious and 140k benign). JSTAP has an accuracy of up to almost 99.5% and outperforms JAST and the closely related work CUJO [16] and ZOZZLE [5]–that we reimplemented–all of which we tested on our dataset.

To further improve our detection accuracy, we combined the predictions of several JSTAP 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. Similarly to JAST, JSTAP is open source and can be used to accurately detect malicious JavaScript code practically and at scale.

4 HIDENOSEEK: Camouflaging Malicious JavaScript in Benign ASTs

We have previously shown that our AST-based detector JAST and our modular and more semantic-oriented detector JSTAP are both very accurate to distinguish benign from malicious JavaScript inputs. Next, we discuss and evaluate the robustness of such static classifiers in an adversarial setting.

4.1 Motivation

Static detectors need to be accurate to not let malicious files through. However, it has been shown that such machine learning-based detectors are susceptible to adversarial attacks, e.g., when an adversary tries to statistically enhance the proportion of benign features [2, 10, 15, 17, 20], to stochastically manipulate samples till they change classification [6, 22], or leverage transferability properties to induce misclassifications [7, 19]. Still, previous work considers a very strong attacker, with insider information (e.g., target model internals, training dataset, or at least assigned classification score), and is tailored to attack one specific detector only. With HIDENOSEEK, though, we explore to what extent we can present a *generic* attack against static malicious JavaScript detectors.

4.2 HIDENOSEEK

Our classifiers JAST and JSTAP are very precise at detecting malicious JavaScript inputs, because they leverage the fact that benign and malicious JavaScript samples generally have a different AST (cf. Figure 1a). Therefore, a generic way of attacking such static systems consists in rewriting malicious JavaScript samples so that they have exactly the same AST as existing benign scripts (cf. Figure 1b) while retaining the initial malicious semantics. We implemented this approach in our fully automated system HIDENOSEEK.

HIDENOSEEK takes a benign and a malicious JavaScript sample as input and first builds their AST, which it enhances with control and data flow information. Then, it looks for identical sub-ASTs between the benign and the malicious graphs (i.e., isomorphic subgraphs). If HIDENOSEEK can find all malicious syntactic structures in benign trees, it replaces the impacted benign sub-ASTs with the syntactically identical malicious ones and adjusts the benign data flows–without changing the AST–to retain the malicious semantics. Finally, HIDENOSEEK generates the code back from the modified AST. This way, HIDENOSEEK crafts samples with the same AST as the original benign inputs while retaining the malicious semantics of the original malicious samples.

4.3 Evasive Sample Generation

To evaluate our attack, we first collected over 120,000 malicious JavaScript samples. After clustering and deobfuscation, we retain 23 malicious payloads, which we refer to as *seeds*. Overall, we could leverage 22 out of 23 seeds to produce over 90,000 malware samples, which exactly reproduce ASTs from Alexa Top 10k. In addition, on average, we can reproduce the syntactic structure of a given Alexa Top 10 web page with 14 different seeds. Similarly to Android in repackaged applications, HIDENOSEEK documents could be presented as the original version for malicious purposes. This attack would be especially hard to spot due to the perfect mapping onto benign ASTs.

4.4 Evaluation Against Real-World Classifiers

We evaluated our camouflage attack in practice, by classifying HIDENOSEEK samples with JAST and JSTAP (including module combination), and our reimplementations of CUJO [16] and ZOZZLE [5].

In a first scenario, we considered that these classifiers were not aware of our attack (i.e., no HIDENOSEEK samples in the training set). As expected, these detectors all misclassify between 99.95 and 100% of our HIDENOSEEK samples as benign. This highlights the success of our attack, which generalizes to multiple static detectors using different categories of features (e.g., tokens, AST, PDG, etc.).

In a second scenario, we leveraged the fact that machine learning-based detectors are able to learn: we added some HIDENOSEEK samples in the training set. This time, only 4.04-8.44% of our HIDENOSEEK samples are misclassified. Still, by design, HIDENOSEEK reproduces existing benign syntax; thus, we subsequently classified the benign files used for our camouflage attack. This time, 88.74-100% of them are misclassified as malicious. As previously, our attack generalizes to multiple categories of static detectors, which are all inept to handle our HIDENOSEEK crafted samples: either they have an extremely high number of false negatives (first scenario) or of false positives (second scenario).

However, for the detectors that do not solely rely on AST, we observe between 0.4 and 11% of the samples that changed classification between the benign and malicious variants. While this is too low to provide a defense mechanism, this may be an interesting avenue for future work. In our thesis, we subsequently discuss potential detection strategies.

5 DOUBLEX: Statically Analyzing Browser Extensions at Scale

Besides *malicious* JavaScript, attackers can also leverage *vulnerable* JavaScript code to perform malicious activities. Browser extensions are a target of choice for malicious actors, as extensions have elevated privileges compared to web pages. For example, to be effective, an ad-blocker needs to read and write web page content or intercept network requests.

This way, while browser extensions are popular to improve user browsing experience, they may also introduce severe security and privacy threats and put their large user base at risk, e.g., leading to universal cross-site scripting.

Prior work mostly focusses on detecting *malicious* extensions [3, 11, 12, 14]. On the contrary, *vulnerable* extensions are more challenging to detect due to their inherently benign intent (as they are not doing anything suspicious). While EmPoWeb [18] focusses on vulnerable extensions, it reports 3,300 / 66,000 extensions as suspicious. Besides the huge manual validation effort, its false-positive rate of 95% renders it inept to be deployed in practice. With DOUBLEX, we address these challenges by proposing a *precise* static analyzer that detects *vulnerable* extensions *at scale*.

5.1 Threat Model

To exploit vulnerable extensions, we consider two attacker scenarios: a Web Attacker and a Confused Deputy. In fact, browser extensions can communicate with web pages (and other extensions) over messages. Therefore, a web page (or another extension) under the control of an attacker can send malicious payloads to a vulnerable extension, tailored to exploit its flaws. This way, an attacker can elevate their privileges to the capabilities of an extension. In particular, an attacker could gain the capability of, e.g., executing arbitrary code in any websites (even without a vulnerability in the websites themselves), making arbitrary cross-origin requests even when a user is logged in, or exfiltrating sensitive user data.

Overall, such vulnerabilities originate from the fact that external actors (i.e., an attacker) can either control the input of security-critical APIs in extensions or receive sensitive user data from a vulnerable extension. Therefore, to detect such flaws, we look for suspicious external data flows in extensions.

5.2 DOUBLEX

To this end, we introduce our static analyzer DOUBLEX. DOUBLEX abstracts the source code of an extension with a graph, including control and data flow, and pointer analysis information. This way, DOUBLEX can handle, e.g., aliased variables or APIs not written in plain text. In addition, DOUBLEX models the intricate message interactions between an extension's components with a *message* flow. It also reports on external messages and flags them as attacker controllable. This way, it can perform a data flow analysis to identify any path between an external actor (i.e., an attacker) and security- or privacy-critical APIs in extensions. Figure 3 presents a schematic depiction of the way DOUBLEX analyzes an extension. In particular, we consider integrity (attacker-controllable data enters a sink) and confidentiality (user sensitive data is exfiltrated) threats in tracking relevant data flows. Finally, DOUBLEX summarizes its findings in a fine-grained data flow report.

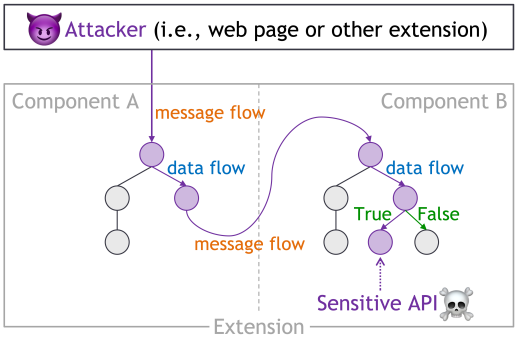


Figure 3: Schematic depiction of the way DOUBLEX processes an extension: 1) for each extension component, DOUBLEX builds their AST enhanced with control (green) and data (blue) flow edges; 2) DOUBLEX models message (orange) interactions within and outside of an extension; 3) DOUBLEX performs a data flow analysis to detect any path (purple) between an attacker and security- or privacy-critical APIs in an extension

5.3 Large-Scale Analysis of Extensions

We evaluated DOUBLEX on 155,000 Chrome extensions, which we collected in 2021. Of those, DOUBLEX reports 309 suspicious data flows (in 278 extensions) between an attacker and the security- and privacy-critical extension APIs we considered. We manually reviewed these reports and confirm that 89% of the suspicious data flows reported can effectively be influenced by external actors. Specifically, we could exploit 209 of these data flows (in 184 extensions) under our threat model. Of those, almost 40% can be exploited by *any* websites. Overall, the 184 vulnerable extensions that DOUBLEX detected impact between 2.4 and 2.9 million users.

To evaluate false negatives (i.e., vulnerable extensions we may have missed), as a best-effort strategy, we evaluated DOUBLEX on the vulnerable extension set from EmPoWeb [18]. Overall, we could detect 93% of known flaws.

5.4 Life Cycle of Vulnerable Extensions

We also evaluated DOUBLEX on over 165,000 Chrome extensions from 2020. This enables us to reason about the life cycle of vulnerable extensions. In particular, 87% of the extensions vulnerable in 2021 were already vulnerable in 2020, despite our disclosure in 2020 and half of those extensions having been updated in between. This finding highlights the need for a system like DOUBLEX to prevent vulnerable extensions from entering the Chrome Web Store in the first place, all the more as they tend to stay in the Store.

Given the high precision (89%) and recall (93%) of DOUBLEX, and its median run-time of 2.5 seconds per extension, we believe that it could be integrated into the vetting

process conducted by Google[2] to detect vulnerable extensions *before* large-scale deployment. In addition, to raise awareness and enable developers to automatically detect such security and privacy threats, we made DOUBLEX publicly available.

6 Discussion and Conclusion

This thesis revolves around studying JavaScript security through static analysis. In particular, we developed advanced methods to uncover JavaScript-based attacks, which would compromise the security and privacy of Internet users. We focussed on two orthogonal threat models, namely *malicious* JavaScript code and *benign-but-buggy* (vulnerable) JavaScript in browser extensions.

First, we designed JAST and JSTAP to detect malicious JavaScript samples at scale. Our fully static learning-based pipelines rest on the AST, enhanced with control and data flow information, to distinguish benign from malicious JavaScript code with an accuracy of 99.5-99.73%. For practical applicability, we made our source code publicly available.

Then, we focussed on the robustness of such static detectors. With HIDENOSEEK, we introduce a *generic* camouflage attack, which consists in rewriting malicious JavaScript samples so that they have exactly the same AST as existing benign scripts. This leads to a no-win situation for static detector operators, who have to choose between either an extremely high number of false positives (88.74-100%) or false negatives (99.95-100%). More generally, we believe that our HIDENOSEEK attack is a good test-bench for machine learning-based detection, as the lessons learned from our study can provide insights into the robustness of other systems in different domains. Naturally, we hope that our work will pave the way for additional research in the malware detection field and lead to the design of fast and more robust systems.

Finally, we considered an orthogonal threat model, namely vulnerable browser extensions. We built DOUBLEX, which reports suspicious external data flows in browser extensions with a high precision (89%) and recall (93%). Overall, we detected 184 Chrome extensions exploitable under our threat model, 87% of which were already vulnerable one year ago. With our approach–that we made publicly available–we hope to increase the awareness of well-intentioned developers toward unsafe programming practices leading to security and/or privacy issues. We believe that integrating DOUBLEX into Chrome vetting system could contribute to detecting such flaws *before* large-scale deployment of the impacted extension; thus, contribute to the protection of end users' security and privacy on the Internet.

References

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

[2] Cao, Y., Pan, X., Chen, Y., and Zhuge, J. JShield: Towards Real-time and Vulnerability-based Detection of Polluted Drive-by Download Attacks. In: *ACSAC*. 2014.

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

[4] chrome. *How long will it take to review my item?* https://developer.chrome.com/webstore/faq#faq-listing-108. Accessed on 2021-12-13.

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

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

[7] 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.

[8] Fass, A. Studying JavaScript Security Through Static Analysis. URL: https://publications.cispa.saarland/3471/7/fass2020thesis.pdf. PhD thesis. Saarland University, 2020.

[9] 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).

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

[11] 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.

[12] 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.

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

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

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

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

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

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

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

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

[2] which currently only targets malicious extensions [4]

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

[22] Xu, W., Qi, Y., and Evans, D. Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. In: *NDSS*. 2016.