UC Irvine
UC Irvine Electronic Theses and Dissertations

Title

Automated Filter Rule Generation for Adblocking

Permalink
https://escholarship.org/uc/item/7d3374cq

Author
Le, Hieu

Publication Date
2023

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Automated Filter Rule Generation for Adblocking

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Hieu Le

Dissertation Committee:
Professor Athina Markopoulou, Chair
Professor Zubair Shafiq
Professor Salma Elmalaki

2023

Chapter 3 © 2021 The Internet Society
Portions of Chapter 4 © 2023 USENIX Association
All other materials © 2023 Hieu Le

DEDICATION

To my parents, who supported me since Day 1 of my academic pursuit.
To my close friends, who kept my spirits high during my PhD journey.

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **x**

LIST OF ALGORITHMS **xiii**

ACKNOWLEDGMENTS **xiv**

VITA **xv**

ABSTRACT OF THE DISSERTATION **xvi**

1 Introduction **1**
1.1 Motivation 1
1.2 Contributions 5
1.2.1 CV-Inspector: Automated Detection of Adblock Circumvention . . 5
1.2.2 AutoFR: Automated Filter Rule Generation for Adblocking 6

2 Background & Related Work **7**
2.1 Web and Adblocking 7
2.1.1 Advertising 7
2.1.2 Filter Rules 9
2.1.3 Machine Learning and Adblocking 11
2.2 Countermeasures against Adblocking 13
2.2.1 Whitelisting 13
2.2.2 Anti-adblocking 14
2.3 Adblocking Beyond the Web 16

3 CV-Inspector: Automated Detection of Adblock Circumvention **18**
3.1 Introduction 18
3.2 Background on Adblock Circumvention 20
3.2.1 Circumvention 21
3.3 State of Anti-Circumvention 24
3.3.1 Filter Rules Overview 24
3.3.2 Analysis of the Anti-circumvention List (ACVL) 25
3.4 CV-Inspector: Design and Implementation 29

3.4.1 Instrumentation and Data Collection . . . 30
3.4.2 Differential Analysis . . . 36
3.4.3 Feature Extraction . . . 39
3.4.4 Ground Truth Labeling . . . 43
3.4.5 The CV-Inspector Classifier . . . 47
3.4.6 Feature Robustness . . . 51
3.4.7 Summary . . . 53
3.5 CV-Inspector: In the Wild Deployment . . . 54
3.5.1 Discovering Circumvention in the Wild . . . 54
3.5.2 Monitoring Circumvention for Sites of Interest . . . 60
3.6 Discussion and Future Directions . . . 62

4 AutoFR: Automated Filter Rule Generation for Adblocking 65
4.1 Introduction . . . 65
4.2 Most Closely Related Work . . . 68
4.3 The AutoFR Framework . . . 69
4.3.1 Filter List Authors' Workflow . . . 70
4.3.2 Reinforcement Learning Formulation . . . 72
4.3.3 The AutoFR Algorithm . . . 78
4.4 AutoFR Implementation . . . 81
4.4.1 Environment . . . 82
4.4.2 Agent . . . 85
4.4.3 Automating Visual Component Detection . . . 86
4.5 Evaluation . . . 88
4.5.1 Filter Rule Evaluation Per-Site . . . 88
4.5.2 AutoFR *vs.* EasyList: Comparing Rules . . . 96
4.5.3 Robustness of AutoFR Filter Rules . . . 101
4.6 Generating Rules Across Multiple Sites . . . 105
4.6.1 Per-site vs. Global Filter Rules . . . 106
4.6.2 Methodologies to Generating Filter Rules . . . 107
4.6.3 Evaluation . . . 111
4.7 AutoFR in a Live Environment (AutoFR-L) . . . 116
4.7.1 AutoFR vs. AutoFR-L . . . 116
4.7.2 AutoFR-L Implementation . . . 118
4.8 Conclusion & Future Directions . . . 120

5 Conclusion 121
5.1 Summary . . . 121
5.2 Perspective . . . 122

Bibliography 125

LIST OF FIGURES

Page

1.1 **The Web Advertising and Tracking Ecosystem.** This ecosystem consists of several key players, from left to right: (1) the users who visit websites using a browser; (2) the publishers who own websites and sell ad locations (slots) to display ads (*i.e.,* the sellers/supply side); (3) the adblockers that block and hide ads and tracking (*i.e.,* PETs); (4) the intermediary companies that provide ad circumvention services (BlockThrough, ExoClick), ad exchanges that connect sellers and buyers (DoubleClick, PubMatic) while tracking users, analytic services to track users (Google Analytics); and (5) the advertisers (Coke and Nike) who buy ad slots to display their ads (*i.e.,* the buyers/demand side). 2

2.1 **Anti-adblocking.** (1) If JS detects that an ad is missing; (2) it shows a popup window asking the user to disable the adblocker, pay for a subscription, or whitelist the site. 15

3.1 **Obfuscation-based Circumvention.** (1) If JS detects that an ad is missing; (2) it sends an obfuscated ad request through a CV server; (3) the server retrieves the new ad from an ad server; (4) the server obfuscates it before sending it back to the browser; (5) JS rebuilds the ad content into DOM elements; and (6) re-injects the ad back onto the page. Compare this workflow with anti-adblocking in Fig. 2.1. 21

3.2 **Anti-circumvention List Over Time.** This shows how filter rules from ABP's ACVL have evolved from May 2018 to May 2020 and categorizes them by filter types. 25

3.3 **Commits by Filter Type.** A boxplot of commit changes from 2018 to 2020 and categorized by filter types for ACVL. The horizontal lines within the boxes represent the median, while the white circles represent the mean. . . . 26

3.4 **Time between Commits.** The time between commits for ACVL is most frequently within 4 minutes while the average is 2.3 hours. 27

3.5 **Tranco-ranking of ACVL.** Sites extracted from ACVL and their corresponding Tranco-ranking. We see that there is low coverage of sites for circumvention from ranking 100k to one million. Note that about half of the sites do not appear in the Tranco top one million list (labeled as 1M+). . . . 28

3.6 **CV-Inspector Workflow.** Given a list of URLs, our crawling script will visit each site four times for: (A) "No Adblocker" and (B) "With Adblocker." With each visit, we collect web requests, DOM mutation events, temporal events (*e.g.*, timestamps and blocked events by the adblocker), and the page source. We take the set difference between the data collected in the two cases, (B)-(A), as websites commonly employ circumvention when an adblocker is on. We use the data to extract most features, train and evaluate our classifier. 30
3.7 **Example of Temporal Features.** We show the number of DOM mutations (spikes) over time for "No Adblocker" and "With Adblocker" (with the corresponding blocked events). We define a cluster of activity as consecutive spikes (no more than one bin apart) and the cluster size as the number of bins that it spans. The top figure shows the "No Adblocker" case, which has 9 clusters with an average cluster size of 8.33. In the middle figure, we show the "With Adblocker" case, which has 22 clusters with an average size of 3.86. In the bottom figure, the dashed vertical lines represent whether blocking events occurred. The majority of blocking happened within the first 12 seconds when compared to the remaining time (*e.g.*, 11 events *vs.* 1 event). 41
3.8 **Example of "Suspicious Content."** The website, *gamer.com.tw*, shows suspicious content on the right sidebar outlined in red. Note that the three small images change between the (a) "No Adblocker" and (b) "With Adblocker" sub-figures. Although the content may look like ads, it could also be benign content related to gaming. Using a browser, we looked at their outgoing URLs and observed that the two smaller images for Tera Awaken and EOS are ads, while the third image links to a first-party page. Since there are still ads displayed in (b) "With Adblocker," we label this example as a positive label. 44
3.9 **Labeling Methodology:** We start with a list of sites from both ACVL and Tranco top-2K, as Candidates for Labeling (CL). We develop an iterative process for prioritizing which (500 in a batch) sites to inspect and label next, then add them to ground truth. We bootstrap a classifier by using outlier detection to find positive labels. In each iteration, we apply the classifier on the remaining sites in CL, sort the sites by decreasing classifier confidence, and inspect and label the 500 sites where the classifier is most confident. Compared to picking randomly 500 sites to label, this heuristic prioritization discovers more positive labels. For example, see Fig. 3.10 between "Iteration Zero" and "Iteration Zero Random." We add the newly labeled samples into our ground truth, retrain our classifier, repeat the process for two more iterations, and declare "Done" when the performance converges, as shown in Fig. 3.10. We combine all labeled data into our Ground Truth (GT) dataset. 46
3.10 **Positive Labels and F1 (per Iteration):** For our ground truth, we show how many positive labels (sites with successful circumvention) were discovered within each iteration. When we compare iteration zero and the randomly chosen iteration zero, we find that our methodology discovers twice as many positive labels. We see that by the end of iteration two, we receive diminishing returns on our classifier performance based on its F1-score. Note that we only find 55 positive labels from the Tranco top-2K overall. 46

3.11 **Top-Features ECDF.** We show empirical CDFs of some of the top features for our classifier. JS path entropy is the most discriminatory feature. 49
3.12 **Discovery vs. Precision.** The trade-off between discovering more circumvention sites (positive instances) within our Tranco-20K (in the wild) dataset *vs.* being correct in the prediction. 56
3.13 **Trade-offs on the Tranco-20K Dataset.** **(a)** Within our ROC curve, we can maximize Youden's J index if we value a high true positive rate (TPR) with a low false positive rate (FPR) for the purpose of discovering sites with successful circumvention. The threshold following these criteria is 0.41, which corresponds to a TPR of 0.85 and FPR of 0.11. **(b)** Within our precision-recall curve, we can find the threshold that corresponds to the optimal F1-score for positive labels. The threshold following these criteria is 0.45, which achieves an F1-score of 0.79. 56

4.1 AutoFR automates the steps taken by FL authors to generate filter rules for a particular site. FL authors can configure the AutoFR parameters but no longer perform the manual work. Once rules are generated by AutoFR, it is up to the FL authors to decide when and how to deploy the rules to end-users. 66
4.2 **(a) Hierarchical Action Space.** A node (filter rule) within the action space has two different edges (*i.e.,* dependencies to other rules): (1) the initiator edge, $\rightarrow$, denotes that the source node initiated requests to the target node; and (2) the finer-grain edge, $\dashrightarrow$, targets a request more specifically, as discussed in Task 4 and Table 4.1. **(b) Site Representation.** We represent a site as counts of visible ads (C_A), images (C_I), and text (C_T), as explained in Sec. 4.3.2.2. Applying a filter rule changes them, by blocking ads (reducing C_A) and/or hiding legitimate content (changing C_I and C_T, thus breakage $\mathcal{B}$). 74
4.3 **AutoFR Example Workflow (Controlled Environment).** INITIALIZE (a–c): (a) spawns $n=10$ docker instances and visits the site until it finishes loading; (b) extracts the outgoing requests from all visits and builds the action space; (c) extracts the raw graphs and annotates them to denote C_A, C_I, and C_T, using JS and Selenium. Once all 10 snapshots are annotated, we run the RL portion of the AUTOFR procedure (steps 1–4). Lastly, AutoFR outputs the rules at step 5, *e.g.,* ||*s.yimg.com/rq/darla/4-10-0/html/r-sf.html*. 81
4.4 **Site Snapshot.** It is a graph that represents how a site is loaded. The nodes represent JS Scripts, HTML nodes (*e.g.,* DIV, IMG, TEXT, IFRAME), and network requests (*e.g.,* URL). "Actor" edges track which source node added or modified a target node. "Requestor" edges denote which nodes initiated a network request. "DOM" edges capture the HTML structure between HTML nodes. Lastly, "Script-used-by" edges track how JS scripts call each other. As described in Sec. 4.4.1, nodes annotated by AutoFR have filled backgrounds, while grayed-out nodes are invisible to the user. 86

4.5 **Site Dynamics.** We consider site dynamics as the unique eSLDs a site contacts. Using our *Full-W09-Dataset*, we show the fraction of unique eSLDs that we collect after every subsequent visit. By the fifth visit, we collected the majority of site dynamics for most sites within our dataset. Besides some outliers, visiting a site 10 times is more than enough to capture site dynamics. 88
4.6 ***Full-W09-Dataset.***: *(a) Action Spaces:* 75% of action graphs have 800 nodes or fewer. AutoFR only needs to explore a fraction of the action space to find effective rules. *(b) Site Snapshots:* 75% of site snapshots contain 10K nodes or fewer. *(c) AutoFR Run–time:* 75% of sites take a minute or less to execute the multi-arm bandit portion of Alg. 1. *(d) Filter Rules:* For 75% of sites, AutoFR generated three filter rules or fewer. 91
4.7 **AutoFR (Top–5K).** All sub-figures exhibit similar patterns. First, the filter rules were able to block ads with minimal breakage for the majority of sites. Thus, the top-right bin (the operating point) is the darkest. Second, there are edge cases for sites with partially blocked ads within the w threshold (right of w line) and sites below the w threshold (left of w line). See Table 4.3, col. 1, 2, and 4, for additional information. 92
4.8 **AutoFR across Different w (Top–5K).** We run AutoFR on *Full-W09-Dataset* using a range of $w \in [0,0.5,1]$ and visualize the effectiveness based on the trade-off of blocking ads *vs.* avoiding breakage. As w increases, there are more sites in operating point. Lower w denotes that the user does not care about breakage, which causes less exploration of the action space for rules that fall in the operating point. 95
4.9 **Comparing AutoFR Rules to EasyList**. Some rules are common and some are unique to each approach. When comparing rules, one must consider the right granularity. 98
4.10 **Δ Site Snapshots between July vs. January 2022.** The differences in site snapshots for nodes, edges, and URLs. A positive change in the x-axis denotes that July had more of the respective factor, while a zero denotes no change. 100
4.11 **Longitudinal Study Every Four Days.** We conduct a longitudinal study of 100 sites over a two-month period. We find that over time, site snapshots will become less similar (*i.e.,* negative Δ Jaccard similarity), denoting that rules are less effective. FL authors can rerun AutoFR on these sites that change more frequently to output effective rules. 102
4.12 **Collateral Damage of Global Rules.** AutoFR rules are generated per-site and can potentially cause breakage when applied to other sites (*i.e.,* treated as a global rule). We report the rules that are unique to AutoFR (*i.e.,* not part of EasyList), ordered by decreasing total collateral damage ($\sum \mathcal{B}$) that they cause to site snapshots within *Full-W09-Dataset*. We can see that most of these rules (93%) cause negligible collateral damage (below 10 on the x-axis). Note that the possible max $\sum \mathcal{B}$ of each rule is the size of the dataset. 104
4.13 **AutoFR** *vs.* **EasyList: Popular Rules** 107

4.14 **Selecting Per-Site Rules into Global Filter Lists.** After creating the per-site AutoFR rules for each site (with $w = 0.9$), we create 10 global filter lists. "Popularity 1" means that a rule is selected into the global list if it was generated in at least one site; "Popularity 10" means that a rule is selected if it was generated for at least 10 sites. Once selected, the rules are now treated as global rules. We apply these global filter lists on our *Full-W09-Dataset* site snapshots and plot the average blocking ads, avoiding breakage, and reward. 108
4.15 **AutoFR-Pop: Top 5K–10K, In the Wild.** We create two filter lists, Fig. 4.15(a) with all rules from *W09-Dataset* and Fig. 4.15(b) that contains rules that were created for ≥ 3 sites. We test them in the wild on the Top–5K to 10K sites (unseen sites) and show their effectiveness along with EasyList (Fig. 4.15(c)). We observe that Fig. 4.15(b) performs better, blocking 8% more ads than Fig. 4.15(a). Table 4.6, col. 1–3, contains additional information. 110
4.16 **AutoFR-L Example Workflow (Live Environment).** INITIALIZE (a–b, Alg. 1): (a) spawns $n = 10$ docker instances and visits the site until it finishes loading; (b) extracts the outgoing requests from all visits and builds the action space. We run the RL portion of AUTOFR procedure (steps 1–4). Lastly, AutoFR outputs the filter rules at step 5, *e.g., ||s.yimg.com/rq/darla/4-10-0/html/r-sf.html*. Note that we do not use AdGraph or site snapshots in this version. . 117

LIST OF TABLES

Page

2.1 Notable terms and their descriptions within this thesis. 8
2.2 Overview of simple (used by EasyList or “EL”) and advanced (used by EL and anti-circumvention filter list or “ACVL”) filter rules. Only the advanced filter rules can stop the execution of JS and take into account the visibility of content when blocking elements. 9
2.3 **Filter Rules Support.** Corresponding to Table 2.2, the approach to adblocking affects the type of filter rules that are supported. “Browser” denotes approaches that render web content using browser extensions, custom browsers, and web views. “Cross-app” approaches utilize local VPNs to decrypt the network traffic of a device to apply filter rules and block ads and tracking. “Cross-device” applies the rules on the DNS traffic for all devices within a particular network. ● = fully supports, ◐ = partially supports, ○ = no support. 16

3.1 Dataset summaries and terminology used throughout the paper. Each of the original datasets is obtained by crawling the corresponding list of sites (and sub-page) and collecting all 4 types of data (web requests, DOM changes, temporal, and page source). 36
3.2 There were 93 features in total in these 4 categories for CV-Inspector. Those marked as “Top” were in the top-10 most important features in Sec. 3.4.5. 38
3.3 **CV-Inspector Cross-validation Results.** Using a Random Forest classifier, 93 features, and 5-fold validation. The label “CV” means successful circumvention and “No CV” means that sites have no CV activity or failed at CV. 48
3.4 **CV-Inspector on the Tranco-20K.** For “No CV” instances, we sample from that predicted set to have a confidence level of 95% with 5% margin of error. 54
3.5 **Circumvention Providers and Approaches**. We show the presence of circumvention providers within the Tranco-20K. We use ● to mean full obfuscation, which means randomized URL components (WR) or deeply nested nonstandard DOM structures for ad (DOM). ◐ denotes partial obfuscation, which means ad resources may be hidden with first-party domain (WR) and ad reinsertion uses simpler DOM structures (DOM). WR = Webrequests, DOM = DOM changes. 58

3.6 **CV-Inspector on the GTP Dataset.** We show the results of applying our classifier on the ~700 sites from our ground truth that also originated from ACVL (Table 3.1). However, this time we collect the data by turning on ACVL as well within our custom ABP extension. For "No CV" instances, we sample from that predicted set to have a confidence level of 95% with a 5% margin of error. 60
3.7 **CV-Inspector on the Adblock Plus Monitoring Dataset.** From a real world dataset used by ABP to monitor circumvention, we apply our classifier and show the results. For "No CV" instances, we sample from that predicted set to have a confidence level of 95% with a 5% margin of error. 60

4.1 **URL-based Filter Rules.** They block requests, listed from coarser to finer-grain: eSLD (effective second-level domain), FQDN (fully qualified domain), With Path (domain and path). Other types of filter rules are provided in Table 2.2. 70
4.2 **AutoFR Top–5K Datasets** 90
4.3 **Results.** We provide additional results to Fig. 4.7. We explain the meaning of each row: (1) the number of sites that are in the operating point (top-right corner of the figures), where filter rules were able to block the majority of ads with minimal breakage; (2) the number of sites that are within w; and (3) the fraction of ads that were blocked across all ads within w. **Confirming via Visual Inspection (In the Wild)* (Sec. 4.5.1): col. 3 is based on a binary evaluation. As it is not simple for a human to count the exact number of missing images and text, we evaluate each site based on whether the rules blocked all ads or not (*i.e.,* $\widehat{C}_A$ is either 0 or 1) and whether they caused breakage or not (*i.e.,* $\mathcal{B}$ is either 0 or 1). For col. 5 (Sec. 4.5.3.1), we repeat the same experiment of col. 2 during July 2022 for a longitudinal study of AutoFR rules. 91
4.4 **Effects of w.** We show how w changes the generated rules for one site. As w increases, some rules will no longer be outputted and new rules may be discovered. While others will become more specific. 95
4.5 **Generating Filter Rules Across Multiple Sites.** We compare different approaches for generating filter rules. We split them into two categories. "Construction" approaches optimize rules during the training process, while the others apply a "post-processing" step on existing per-site rules. We use AutoFR (row 1) as our baseline. The column "Generalizes" denotes whether the approach can deal with unseen sites. Efficiency provides empirical estimates of each approach in minutes. Square brackets [ ] denote that parallalization can be used to remove n, *e.g.,* $1.6\times n \rightarrow 1.6$. 105
4.6 **AutoFR-Pop** (Top 5K–10K, In the Wild). We evaluate AutoFR-Pop on unseen sites. See Fig. 4.15. 110

4.7 **Generating Rules across Multiple Sites (using Site Snapshots).** We leverage our *W09-Dataset* to evaluate our methodologies (from Table 4.5) for generating filter rules that can be applied across multiple sites. We provide recommendations by comparing the approaches based on performance on the known sites (training set), how well they generalize to unseen sites (test set), and their empirical efficiency and maintainability (over time). We use the following criteria for each methodology from col. 7–8: ○ = 30+% from the baseline, ⊙ = 30%, ◐ = 20%, ◉ = 10%, ● = same as baseline. On the other hand, "Best Overall" treats ● = 1. We then sum up the scores of each row from col. 7–10 and take their averages. *e.g.,* row 1 has ◉ = 3/4. . 112

LIST OF ALGORITHMS

Page

1 AutoFR Algorithm 80

ACKNOWLEDGMENTS

To my advisor, Athina Markopoulou, I would like to extend my deepest gratitude. You motivated me to pursue higher education and was instrumental in my publication success. Thank you for taking a chance on a student who had no prior experience with research and investing your time to forge him into a published author. Moreover, thank you for the continuous funding, which gave me the privilege to focus on research without disruption. And for providing well-equipped labs, which made for an outstanding work environment. You have been a true role model. Your patience, guidance, and support empower me to pursue academia further.

To my committee members, Zubair Shafiq and Salma Elmalaki, it has been a pleasure working with and learning from you both. To Zubair Shafiq, thank you for steering me toward the subject of advertising and tracking. It has proven to be an engaging topic to research. You have been a trustworthy source of support and advice. To Salma Elmalaki, thank you for guiding me during my favorite research project, AutoFR. Thank you for your consistent availability and patience while I learned reinforcement learning. To all three professors, I am sincerely grateful for giving me the space to make mistakes and to grow through them.

Thank you to my collaborators: Anastasia Shuba, Janus Varmarken, Rahmadi Trimananda, Hao Cui, Janice Ho, Qingchuan Yang, Yiyu Qian, and Stelios Stavroulakis. I especially acknowledge Anastasia Shuba for inspiring me to do my PhD and introducing me to Athina. To Janus Varmarken, I appreciate your mentorship during our first research paper together.

I thank my friends and fellow EECS cohort members for assisting me in balancing my research commitments with social events, particularly: Huy Nguyen, Kelly Fukomoto, David Lim, Sean Kocol, Kimberly Kocol, Max Nanasy, Beverly Quon, Robert Marosi, Joann Chen, Floranne Ellington, Nathan Furman, and Nilab Ismailoglu. In addition, I am grateful for the ProperData members who provided support and advice, particularly: Umar Iqbal, Marilyne Tamayo, Tianyue Chu, and Devriş İşler.

Thank you to Gary De La Cruz, who supported me during all five-plus years of my PhD. I genuinely would not be able to survive this journey without you.

Thank you to the adblocking community, including Oleksandr Paraska, Uwe Bernitt, Shwetank Dixit, Ryan Brown, Arthur Kawa, and Peter Lowe, who provided valuable insights into ad circumvention and the human process of creating filter rules. Thank you to eyeo for inviting me to showcase my work at the Ad-Filtering Dev Summit for the last four years.

This thesis was partially supported by NSF Awards 1956393, 1939237 and 1815666.

Reprint Notice

Portions of this dissertation are reprints of, or largely based on, the materials in [85, 84], used with permission from the Internet Society and USENIX Association, respectively.

VITA

Hieu Le

EDUCATION

Doctor of Philosophy in Electrical and Computer Engineering **2023**
University of California, Irvine *Irvine, CA, USA*

Master of Science in Computer Engineering **2021**
University of California, Irvine *Irvine, CA, USA*

Bachelor of Computer Science and Engineering **2009**
University of California, Irvine *Irvine, CA, USA*

EXPERIENCE

Graduate Research Assistant **2018–2023**
University of California, Irvine *Irvine, CA, USA*

Curriculum Co-Lead **June 2023**
ProperData: Privacy and IoT Research and Exploration Workshop *Irvine, CA, USA*

Anti-circumvention Intern **Summer 2021**
eyeo *Irvine, CA, USA*

REFEREED CONFERENCE PUBLICATIONS

H. Le, S. Elmalaki, A. Markopoulou, and Z. Shafiq. "AutoFR: Automated Filter Rule Generation for Adblocking." *In Proc. of the 32nd USENIX Conference on Security Symposium (SEC 2023).* Anaheim CA, Aug. 2023.

R. Trimananda, **H. Le**, H. Cui, J. Ho, A. Shuba, A. Markopoulou. "OVRseen: Auditing Network Traffic and Privacy Policies in Oculus VR." *In Proc. of the 31nd USENIX Conference on Security Symposium (SEC 2022).* Boston MA, Aug. 2022.

H. Le, A. Markopoulou, and Z. Shafiq. "CV-Inspector: Towards Automating Detection of Adblock Circumvention." *In Network and Distributed System Security Symposium (NDSS).* Virtual, Feb. 2021.

J. Varmarken, **H. Le**, A. Shuba, A. Markopoulou, and Z. Shafiq. "The TV is Smart and Full of Trackers: Measuring Smart TV Advertising & Tracking." *In Proc. of Privacy Enhancing Technologies Symposium (PETS/PoPETS).* Issue 2, pp. 129-154, Virtual, May 2020.

ABSTRACT OF THE DISSERTATION

Automated Filter Rule Generation for Adblocking

By

Hieu Le

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2023

Professor Athina Markopoulou, Chair

Advertising is prevalent across different platforms, especially on the web. To combat this, millions of users rely on privacy-enhancing technologies, such as adblockers. They are powered by filter rules, which are string-based patterns that block and hide advertising and tracking. However, these filter rules are manually curated and continuously maintained by human experts. This is further exacerbated by technical reasons, as advertising and tracking are employed in different ways across websites. In addition, for economic reasons, such as when websites and advertisers attempt to circumvent adblockers to earn revenue through advertising, effectively causing an arms race.

In this thesis, we examine this arms race and develop methodologies and frameworks to improve adblockers in terms of automation, scalability, and robustness. To achieve these goals, we make the following contributions. First, we examine the human effort necessary to update filter rules that combat adblock circumvention by conducting a longitudinal analysis. To detect circumvention, we build CV-Inspector, a machine-learning approach that leverages differential analysis to capture features of circumvention across HTTP and HTML DOM modalities. CV-Inspector reduces the human maintenance cost for filter rules, as it removes the manual monitoring of websites for circumvention. Our second contribution studies the problem of filter rule generation from scratch by developing a reinforcement learning

framework called AutoFR. Our formulation enables us to automate the human process of filter rule creation and maintenance. Notably, the AutoFR framework is tunable (*e.g.,* users can explore different reward functions) and applicable beyond ads and the web (*e.g.,* generate rules that block tracking for mobile apps). We demonstrate that both CV-Inspector and AutoFR are effective in a controlled setting and in the wild, *i.e.,* when applied to real websites, even for those we have not trained on. We envision our tools and methodologies will be useful to the adblocking community to improve and automate the creation and maintenance of adblocking filter rules.

Chapter 1

Introduction

1.1 Motivation

Advertising is the primary business model that companies depend upon to thrive within online platforms. Those who sell products and services can reach a larger audience through advertising. Others, such as social media platforms, leverage their large user base to display ads and earn revenue. The profitability of ads hinges on knowing their audience well. This incentivizes the adoption of techniques such as cookie syncing and fingerprinting to track users and infer details, such as their demographics, locations, and behaviors. Thus, tracking enables the collection of data and the creation of user profiles, which facilitates targeted advertising.

Advertising and tracking are interconnected and impact millions of consumers worldwide. Both are prevalent across platforms like the web, mobile, smart TVs, voice assistants, and soon-to-be on extended reality devices. This has brought some benefits for users; they can enjoy free services, such as movies, games, and news content, with interspersed ads. However, this creates a trade-off with user privacy — the data collection practices and the inner workings of the advertising and tracking ecosystems are opaque to users. In addition, it comes at

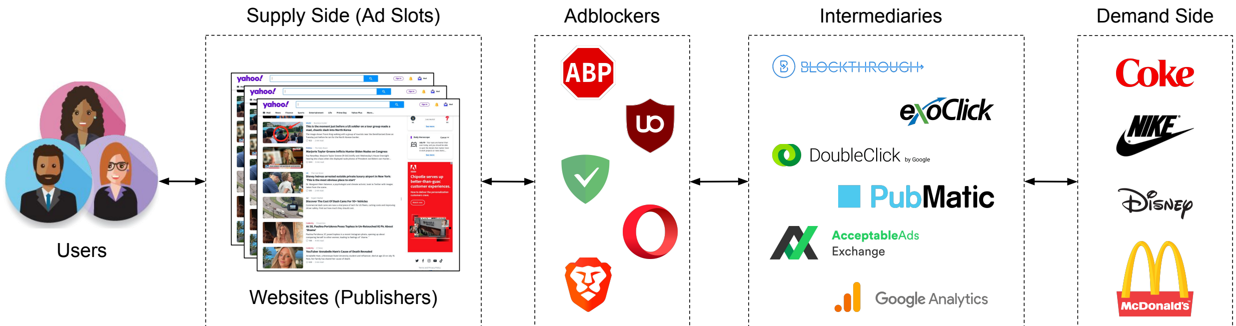


Figure 1.1: **The Web Advertising and Tracking Ecosystem.** This ecosystem consists of several key players, from left to right: (1) the users who visit websites using a browser; (2) the publishers who own websites and sell ad locations (slots) to display ads (*i.e.*, the sellers/supply side); (3) the adblockers that block and hide ads and tracking (*i.e.*, PETs); (4) the intermediary companies that provide ad circumvention services (BlockThrough, ExoClick), ad exchanges that connect sellers and buyers (DoubleClick, PubMatic) while tracking users, analytic services to track users (Google Analytics); and (5) the advertisers (Coke and Nike) who buy ad slots to display their ads (*i.e.*, the buyers/demand side).

a cost to the user experience. Advertising can slow down websites, display disturbing images, disrupt users with frequent popups, and unsettle users with re-targeted ads across websites.

There are several ways to minimize these costs. First, alter the business model of ads to another. For example, utilize a subscription-based model to earn revenue. This is already present in services such as Netflix (movies), PlayStation Plus (games), and the New York Times (news). Another possible business model is to pay users to view ads. We see this emerging model in Brave Rewards [37], which monetarily rewards users for viewing ads and lets them spend it on publishers of their choice. Second, introduce privacy regulations, such as the California Consumer Privacy Act (CCPA) [39] and General Data Protection Regulation (GDPR) [136]. They require companies to report how user data is collected, shared, and sold, and for which purposes within privacy policies. As a result, this improves the transparency of how data is collected and utilized for advertising. Third, design technical ad standards such as the Authorized Digital Sellers (ads.txt) from the Interactive Advertising Bureau (IAB) [81]. Websites can declare who is permitted to sell their ad inventory. Or the Acceptable Ads Standard from the Acceptable Ads Committee (AAC) [3], which dic-

tates the format, size, and frequency of ads to improve the user experience. Fourth, develop privacy-enhancing technologies (PETs), such as adblockers, to block, hide, and obfuscate advertising and tracking. This proactive but hard-line approach prevents companies from earning ad revenue and encourages them to adopt alternatives that do not impact user privacy and experience. One such alternative is the Privacy Sandbox, proposed by Google [63]. It is an initiative to develop and apply privacy-preserving approaches during the collection and processing of data, such as differential privacy and on-device processing.

This dissertation focuses on web adblockers within browsers and the arms race between key players that want to serve ads to earn revenue *vs.* those who want to block them. Fig. 1.1 illustrates an overview of this ecosystem. Adblockers are the most widely adopted type of privacy-enhancing technology and have been installed by millions worldwide. They can be browser extensions (*e.g.,* Adblock Plus [10], AdGuard [13], uBlock Origin [135]) and browsers (*e.g.,* Brave, Opera), which are the most prevalent way to access the web[1]. Their efficacy depends on filter rules, string-based patterns that can match with HTTP requests (*e.g.,* *||example.com/ads/*) or HTML elements (*e.g.,* *||example.com##.ad-container*) to block and hide advertising and tracking. They are practical beyond this purpose, such as blocking phishing and malware. These filter rules are contained within filter lists, often containing thousands of individual rules. By 2022, there were over 2K filter lists supported by 44 different software tools for 42 diverse purposes [25]. However, filter rules are manually curated and continuously maintained by human experts who adblocking companies employ (*i.e.,* filter list authors) or by crowd-sourcing endeavors. The magnitude of this endeavor is exacerbated by sites that change naturally over time and by companies who want to evade adblockers. Thus, we study the challenges to improving the scalability of adblockers and to maintaining their efficacy. To achieve this, we develop methodologies and frameworks that minimize human involvement in creating and updating filter rules.

[1] They come in other forms, such as mobile applications (*e.g.,* AntMonitor [121]) and standalone applications (*e.g.,* Pi-hole [108], AdGuard Home [12]).

First, we investigate the advanced techniques websites and advertisers utilize to circumvent adblockers. This includes techniques such as randomizing URL components, which evade filter rules due to their static nature. To reduce human intervention, we train a machine-learning classifier using HTTP, HTML, and CSS modalities to detect when a website has successfully circumvented the adblocker. This simplifies the process of filter rule maintenance by notifying filter list authors when to update filter rules automatically.

Second, we examine the fundamental problem of creating filter rules from scratch and focus on blocking ads. Blocking ads is easy — break the whole website. It becomes challenging when one has to consider breakage as well. A second difficulty is removing the reliance on existing filter rules manually maintained by human experts. Prior work depends on these rules to label their ground truth to train machine-learning classifiers that predict whether an HTTP request is related to advertising and tracking. Thus, we develop a reinforcement learning framework capable of generating filter rules while considering visual breakage for a website without needing pre-existing rules.

Third, we apply our knowledge of reinforcement learning and automated systems to develop a framework to audit social platforms that rely on recommendation systems. These algorithms assist users in finding new and relevant content through personalization within "For You" pages. However, they can also serve unvetted harmful and hateful content. Thus, our framework aims to audit recommendation systems to inform non-profits, researchers, and policymakers of undesired content that should either be removed or suppressed; and ultimately, to protect users from viewing them.

This dissertation comprehensively addresses the major pain-points of filter rule generation and maintenance. When put into practice, our work can improve the robustness and scalability of adblockers, which ultimately gives users more control over their privacy on the web. Lastly, our methodologies and frameworks are extendable to generate filter rules for tracking and for other platforms where there are fewer available privacy-enhancing tools for users.

1.2 Contributions

The outline and contributions of this thesis are as follows:

1.2.1 CV-Inspector: Automated Detection of Adblock Circumvention

In Chapter 3, we are interested in studying the escalated arms race between publishers, advertisers, and adblockers over the last few years. An entirely new ecosystem of circumvention (CV) services has recently emerged that aims to bypass adblockers by obfuscating site content, making it difficult for adblocking filter lists to distinguish between ads and functional content. In this chapter, we investigate recent anti-circumvention efforts by the adblocking community that leverage custom filter lists. In particular, we analyze the anti-circumvention filter list (ACVL), which supports advanced filter rules with enriched syntax and capabilities designed specifically to counter circumvention. We show that keeping ACVL rules up-to-date requires expert list curators to continuously monitor sites known to employ CV services and to discover new such sites in the wild — both tasks require considerable manual effort. To help automate and scale ACVL curation, we develop CV-Inspector, a machine-learning approach for automatically detecting adblock circumvention using differential execution analysis. We show that CV-Inspector achieves 93% accuracy in detecting sites that successfully circumvent adblockers. We deploy it on Top–20K sites to discover the sites that employ circumvention in the wild. We further apply CV-Inspector to a list of sites that are known to utilize circumvention and are closely monitored by ACVL authors. We demonstrate that it reduces the human labeling effort by 98%, which removes a major bottleneck for ACVL authors. This chapter presents the first large-scale study of the state of the adblock circumvention arms race and makes an important step towards automating anti-CV efforts. This work was also published in [85].

1.2.2 AutoFR: Automated Filter Rule Generation for Adblocking

In Chapter 4, we propose a framework to generate filter rules from scratch. Adblocking relies on filter lists, which are manually curated and maintained by a community of filter list authors. Filter list curation is a laborious process that does not scale well to a large number of sites or over time. We introduce AutoFR, a reinforcement learning framework to fully automate the process of filter rule creation and evaluation for sites of interest. We design an algorithm based on multi-arm bandits to generate filter rules that block ads while controlling the trade-off between blocking ads and avoiding visual breakage. We test AutoFR on thousands of sites and we show that it is efficient: it takes only a few minutes to generate filter rules for a site of interest. AutoFR is effective: it optimizes filter rules for a particular site that can block 86% of the ads, as compared to 87% by EasyList, while achieving comparable visual breakage. Using AutoFR as a building block, we devise three methodologies that generate filter rules across sites based on: (1) a modified version of AutoFR, (2) rule popularity, and (3) site similarity. We conduct an in-depth comparative analysis of these approaches by considering their effectiveness, empirical efficiency, and maintainability over time. We demonstrate that some of them can generalize well to new sites in both controlled and live settings. We envision that AutoFR can assist the adblocking community in filter rule generation at scale. This work was also published in [84].

Chapter 2

Background & Related Work

In this chapter, we detail our background and related work. Sec. 2.1 provides information about the advertising ecosystem and adblockers; *e.g.,* filter rules power adblockers and how machine-learning approaches fall short in replacing rules. Sec. 2.2 describes how websites and advertisers have responded to adblockers; *e.g.,* how they try to circumvent adblockers. Sec. 2.3 discusses how adblocking can be applied beyond the web, such as for mobile and smart TV devices. Table 2.1 summarizes notable terms used throughout this thesis.

2.1 Web and Adblocking

2.1.1 Advertising

Advertising is a multi-billion dollar industry and consists of several key players, as explained in Fig. 1.1. To be effective, relevant ads are displayed to users to increase the chances of the user clicking on the ad to purchase a product or service. On the web, advertising can be personalized to users using two main approaches. The first is contextual-based advertising. As the name denotes, this personalizes the ad based on the current context of the user, such

Term	References and Description
Countermeasures to Adblockers	
Whitelisting	Sec. 2.2.1: Allow ads if they conform to certain standards
Anti-adblocking	Sec. 2.2.2, Fig. 2.1: Ask users to disable adblocker or pay for site content
Cloaking-based Circumvention	Sec. 3.2.1: Cloaking approaches that disguise themselves, *e.g.,* sending ad requests using WebSockets and hiding third-party request as first-party with DNS CNAME
Obfuscation-based Circumvention	Sec. 3.2.1, Fig. 3.1: Obfuscation approaches to re-inject ads back by randomizing URL components or HTML ad structures
Filter Rules (Table 2.2)	
Per-site Rules	Sec. 2.1.2: Optimized and applicable for known sites
Global Rules	Sec. 2.1.2: Optimized for known sites, applicable to any sites
Known Sites	Sec. 2.1.2: Given sites during construction of rules (*i.e.,* training set)
Unseen Sites	Sec. 2.1.2: Sites not known during the construction of rules (*i.e.,* test set, generalize)
Visual Breakage	Sec. 2.1.2, Eq. (4.2): Missing legitimate content ($\mathcal{B}$), such as images and text
Collateral Damage	Sec. 2.1.2, Sec. 4.6.1: Sum of unintended breakage ($\sum\mathcal{B}$) on unseen sites
Rule Popularity	Sec. 4.6.2.2: Number of sites that generate a specific rule
Effectiveness of Filter Rules (Sec. 4.3.2.2)	
w threshold	Sec. 4.3.2.2: Value that user tunes to express acceptable breakage
Bad Rules	Eq. (4.3a): Rules that do not help in blocking ads
Potential Rules	Eq. (4.3b): Rules that block ads but cause breakage beyond w
Good Rules	Eq. (4.3c): Rules that block some ads and cause breakage within w
Filter Rule Generation Challenges	
Blocking Ads	Sec. 2.1, 4.3.2.2, Eq. (4.3): Block some or majority of ads
Avoiding Breakage	Sec. 4.3.2.2, 4.6, Eq. (4.3): Minimize breakage and/or collateral damage
Performance	Sec. 4.5, 4.6: Block ads with (visual) breakage within w on known/unseen sites
Efficiency	Sec. 4.5.1, 4.6: Scale across thousands of sites
Robust Rules	Sec. 4.5.3: Generate rules that perform well over time
Maintainability	Sec. 4.6: Fast when updating rules or dealing with unseen sites
Filter Rule Generation Approaches (Sec 4.6.2, Table 4.5)	
AutoFR	Sec. 4.3: Generates per-site URL-based rules optimized for known sites
AutoFR-Global	Sec. 4.6.2.1: Extending AutoFR for global filter rules
AutoFR-Pop	Sec. 4.6.2.2: Using popularity to select per-site rules as global rules
AutoFR-Sim	Sec. 4.6.2.3: Applying per-site rules from similar sites (*e.g.,* common eSLDs)

Table 2.1: Notable terms and their descriptions within this thesis.

as the content of the website that the user visits or the search term that the user utilizes. For instance, ads promoting athletic shoes may be shown to users who search for "running shoes." Note that this may consider other information, such as the device or location of the user. The second is targeted advertising, which infers information about the user based on their behavior as they browse the web [129]. This enables companies to build intricate user profiles that consider multiple interests (*e.g.,* shopping, travel), life stages (*e.g.,* home ownership, marital

<table>
<tr><th>Filter Types</th><th>EL</th><th>ACVL</th><th>Example</th><th>Purpose</th></tr>
<tr><td>Web Request Blocking (hostname)</td><td>✓</td><td>✓</td><td>||a.com^</td><td>Blocks requests matching hostname</td></tr>
<tr><td>Web Request Blocking (URL-based)</td><td>✓</td><td>✓</td><td>||a.com/ads/*/images$script</td><td>Blocks web requests matching domain, path and script type</td></tr>
<tr><td>Web Request Blocking (Per-site)</td><td>✓</td><td>✓</td><td>||a.com/ads/$domain=cnn.com</td><td>Blocks web requests matching domain and path for only cnn.com</td></tr>
<tr><td>Element Hiding</td><td>✓</td><td>✓</td><td>||a.com##.ad-container</td><td>Hides all elements matching class name</td></tr>
<tr><td>Advanced JavaScript Abortion</td><td>✗</td><td>✓</td><td>||a.com#$#abort-on-property-read EX, ||a.com#$#abort-on-property-write EX</td><td>Stops JS execution from reading or writing to window.EX</td></tr>
<tr><td>Advanced Element Hiding</td><td>✗</td><td>✓</td><td>||a.com#$#hide-if-contains-visible-text /Sponsor/</td><td>Hides all elements containing Sponsor text</td></tr>
</table>

Table 2.2: Overview of simple (used by EasyList or "EL") and advanced (used by EL and anti-circumvention filter list or "ACVL") filter rules. Only the advanced filter rules can stop the execution of JS and take into account the visibility of content when blocking elements.

status, education), and purchasing behavior (*e.g.,* loyal or first-time customers, people who abandon their shopping carts). One well-studied platform is the Oracle Data Cloud Register [105], which aggregates user data based on cookies and ad bid values to assign labels that describe a user [28, 148]. This registry contains thousands of labels, such as "Health, Beauty > Style > Dieting > Weight Loss" [50]. To combat this, users install adblockers to both stop seeing ads and reduce the information that online companies collect and infer about users.

2.1.2 Filter Rules

On the web, adblockers come in the form of browser extensions (*e.g.,* Adblock Plus [10], AdGuard [13], uBlock Origin [135]) or integrated directly into browsers (*e.g.,* Brave, Opera). The number of web users who use some form of adblocking now exceeds 42% [24]. The first adblocker in 2002, a Firefox extension, allowed users to specify custom filter rules to block

resources (e.g., images) from a particular domain or URL path [97]. There are different types of filter rules, shown in Table 2.2. The most popular type is URL-based filter rules, which block network requests to provide performance and privacy benefits [125]. Other types of filter rules are element-hiding rules (hide HTML elements) and JS-based rules (stop JS execution). Chapter 3 focuses on advanced rules to combat adblock circumvention.

Per-site *vs.* **Global Rules.** There are two broad types of filter rules that describe how they can be applied. First, there are "per-site" rules, which are restricted to trigger for particular sites. They are denoted with the "$domain" option. Second, there are "global" rules, which can trigger for any site. Examples of both per-site and global rules are provided in Table 2.2. Popular filter lists support per-site and global rules; they contain mostly global rules. Chapter 4 develops approaches to generating both per-site and global URL-based rules.

Known *vs.* **Unseen Sites.** Per-site and global rules are expected to work well on "known" sites, *i.e.,* given sites that the rules are being optimized for during their construction. While per-site rules can be triggered for their known sites, global rules, on the other hand, can also trigger for other "unseen" sites, *i.e.,* sites that the rules were not optimized for. Chapter 4 evaluates the performance of per-site and global filter rules on both known and unseen sites.

Breakage and Collateral Damage. All types of filter rules have the potential to cause visual (*e.g.,* missing legitimate content like images and text) and/or functionality breakage (*e.g.,* breaking infinite scrolling, navigation, form submissions) to sites. If the sites are known during the construction of rules, then this potential for breakage can be minimized (*i.e.,* optimized for). However, rules have the potential to cause unintended breakage when applied to unseen sites, we call this "collateral damage". In Sec. 4.3, we develop a framework to generate per-site rules that consider visual breakage for a known site. We explore the potential for collateral damage when applying rules on unseen sites in Sec. 4.6.1.

Filter Lists (FL). Since it is non-trivial for lay web users to create filter rules, several efforts

were established to curate rules for the broader adblocking community. Specifically, rules are curated by filter list (FL) authors based on informal crowd-sourced feedback from users of adblocking tools. There is now a rich ecosystem of thousands of different filter lists focused on blocking ads, trackers, malware, and other unwanted web resources. EasyList [54] is the most widely used adblocking filter list. Started in 2005 by Rick Petnel, it is now maintained by a small set of FL authors and has 22 language-specific versions. An active EasyList community provides feedback to FL authors on its official forum and GitHub. There are other filter lists, such as the anti-circumvention filter list (ACVL) [8], that support more advanced rules and are further discussed in Chapter 3.

Filter List Curation (and Challenges). The research community has looked into the filter list curation process to investigate its effectiveness and pain-points [125, 85, 138, 19]. Snyder *et al.* [125] studied EasyList's evolution and showed that it needs to be frequently updated (median update interval of 1.12 hours) because of the dynamic nature of online advertising and efforts from advertisers to evade filter rules (*e.g.,* adblock circumvention discussed in Chapter 3). They found that it has grown significantly over the years, with 124K+ rule additions and 52K+ rule deletions over the last decade. Alrizah *et al.* [19] showed that EasyList's curation, despite extensive input from the community, is prone to errors that result in missed ads (false negatives) and over-blocking of legitimate content (false positives). Thus, there are challenges in finding effective rules that block ads while minimizing breakage. They concluded that most errors in EasyList can be attributed to mistakes by FL authors. We elaborate further on the challenges of filter rule generation in Sec. 4.3.1 and maintenance in Sec. 3.3.2 and 4.5.3.

2.1.3 Machine Learning and Adblocking

Machine Learning for Adblocking. Motivated by challenges in creating and maintaining filter lists, and listed in Table 2.1, prior work has explored using machine learning (ML) to

assist with filter list curation or replace it altogether. To assist filter list curation, prior work developed ML models to automatically generate filter rules for blocking ads [30, 66, 124]. Bhagavatula *et al.* [30] trained supervised ML classifiers to detect advertising URLs. Similarly, Gugelmann *et al.* [66] trained supervised ML classifiers to detect advertising and tracking domains. Sjosten *et al.* [124] is the closest related to our work. First, they trained a hybrid perceptual and web execution classifier to detect ad images [35]. Second, they generated adblocking filter rules by first identifying the URL of the script responsible for retrieving the ad and then simply using the effective second-level domain (eSLD) and path information of the script as a rule (similar to Table 4.1 row 3). We found that 99% of rules that they open-sourced had paths. However, this overreliance on rules with paths makes them brittle and easily evaded with minor changes [85]. Furthermore, the design of these rules did not automatically consider potential breakage. Another line of prior work, instead of generating filter rules, trained ML models to automatically detect and block ads [73, 122, 146, 2, 127, 7]. AdGraph [73], WebGraph [122], and WTAGraph [146] represent web page execution information as a graph and then train classifiers to detect advertising resources. Ad Highlighter [127], Sentinel [7], and PERCIVAL [2] use computer vision techniques to detect ad images. These efforts do not generate filter rules but instead attempt to replace filter lists altogether.

While promising, existing ML-based approaches have not seen any adoption by adblocking tools. Our discussions with the adblocking community have revealed a healthy skepticism of replacing filter lists with ML models due to performance, reliability, and explainability concerns. On the performance front, the overheads of feature instrumentation and running ML pipelines at run-time are non-trivial and almost negate the performance benefits of adblocking [93]. On the reliability front, concerns about the accuracy and brittleness of ML models in the wild [7, 124, 2], combined with a lack of explainability [130], have hampered their adoption. In short, it seems unlikely that filter lists will be replaced by ML models any time soon, and filter rules remain crucial for adblocking tools.

Machine Learning Assisted Filter List Curation. There is, however, potential in using ML-based approaches to assist with *maintenance* of filter lists. For example, Brave [124], Adblock Plus [7], and the research community [85] have been using ML models to assist FL authors in prioritizing filter rule updates. However, they have two main limitations. First, they rely on filter lists, such as EasyList, for training their supervised ML models causing a *circular dependency*: a supervised model is only as good as the ground-truth data it is trained on. This also means that the adblocking community has to continue maintaining both ML models as well as filter lists. Second, existing ML approaches do not explicitly consider the trade-off between blocking ads and avoiding breakage. An over-aggressive adblocking approach might block all ads on a site but may block legitimate content at the same time. It is essential to control this trade-off for real-world deployment. In summary, a deployable ML-based adblocking approach should be able to generate filter rules without relying on existing filter lists for training, while also providing control to navigate the trade-off between blocking ads and avoiding breakage. To the best of our knowledge, AutoFR is the only system that can generate and evaluate filter rules automatically (without relying on humans) and from scratch (without relying on existing filter lists).

2.2 Countermeasures against Adblocking

Publishers, companies that own websites, employ various countermeasures against adblocking. They can be divided into three categories: whitelisting (Sec. 2.2.1), anti-adblocking (Sec. 2.2.2), and circumvention (Sec. 3.2.1). We describe the first two categories here and elaborate on circumvention in Chapter 3.

2.2.1 Whitelisting

Some adblockers allow whitelisting of ads if they conform to certain standards. The Acceptable Ads program [4] whitelists ads if they are not intrusive based on their placement,

distinction, and size. Adblock Plus (ABP) and a few other adblockers currently implement the Acceptable Ads based whitelist. The Better Ads Standard [29], by the Coalition for Better Ads, prohibits a narrower set of intrusive ad types, such as pop-up ads and large sticky ads. Google's Chrome browser blocks ads on sites that do not comply with the Better Ads Standard, and whitelists ads on the remaining sites [46]. However, whitelisting is not a silver bullet for publishers. First, it is not supported by many popular adblockers such as uBlock Origin and the Brave Browser. Second, some adblockers, such as ABP, require large publishers to pay a fee to be whitelisted. Publishers may also have to pay a fee to ad exchanges, such as the Acceptable Ads Exchange, to serve acceptable ads.

Prior work has investigated the evolution and impact of ad whitelisting. Walls *et al.* [139] studied the growth of the Acceptable Ads whitelist over the years and showed that it covers a majority of the popular sites. They also reported that the whitelisting process is flawed because "acceptable" ads are often disliked by users due to their intrusiveness and misleading resemblance to page content. In fact, the whitelisting of deceptive ads from content recommendation networks such as Taboola and Outbrain [27] has been quite controversial [6]. Pujol *et al.* [113] showed that most ABP users do not opt-out of the Acceptable Ads whitelist despite these issues. Merzdovnik *et al.* [92] showed that ABP blocked the least amount of ads as compared to other adblocking tools because of whitelisting.

2.2.2 Anti-adblocking

Many publishers deploy anti-adblockers that use client-side JavaScript (JS) to detect adblockers based on whether ads are missing. Fig. 2.1 illustrates the workflow of anti-adblocking. The logic is implemented by a client-side JS that detects whether an ad is missing by measuring the ad's display properties or other dimensions. Then, it displays a popup either warning users to disable their adblockers or a paywall asking them to sign-up for paid subscriptions.

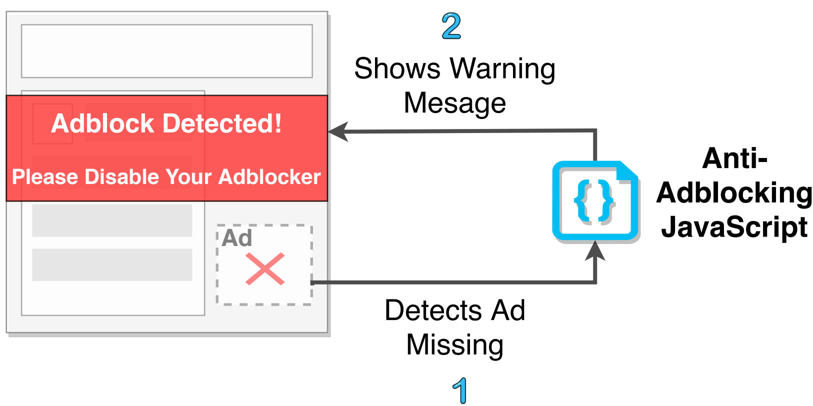


Figure 2.1: **Anti-adblocking.** (1) If JS detects that an ad is missing; (2) it shows a popup window asking the user to disable the adblocker, pay for a subscription, or whitelist the site.

Third-party anti-adblocking services [107, 31, 70] are used by many news publishers such as the Washington Post and Forbes. Nithyanand *et al.* [102] manually analyzed JS snippets to characterize anti-adblockers. Mughees *et al.* [100] trained a machine learning classifier to detect anti-adblockers based on HTML DOM changes. These early studies showed that hundreds of sites had started deploying anti-adblockers.

Adblockers counter anti-adblockers using specialized filter lists that use the same syntax as the standard EL. These filter rules either trick the detection logic of anti-adblockers by allowing baits or hide the warning message shown by anti-adblockers after detection. Iqbal *et al.* [72] studied the coverage of these filter lists (*e.g.*,, Adblock Warning Removal List) against anti-adblocking. They showed that these filter lists are often slow in adding suitable rules by several weeks or sometimes even months. They further trained a machine learning classifier to detect anti-adblocking JS using static analysis. Zhu *et al.* [149] proposed a dynamic differential analysis approach to detect and disrupt anti-adblockers. The countermeasures above have proven reasonably successful against anti-adblockers. Moreover, the warning messages shown by anti-adblockers have proven to be of limited benefit [44, 117]. About three-quarters of surveyed users said that they would simply leave the site instead of disabling their adblocker [107].

Filter Types	**Browser**	**Cross-app**	**Cross-device**
Web Request Blocking (hostname)	●	●	●
Web Request Blocking (URL-based)	●	◐	○
Web Request Blocking (Per-site or Per-app)	●	◐	○
Element Hiding	●	◐	○
Advanced JavaScript Abortion	●	◐	○
Advanced Element Hiding	●	◐	○

Table 2.3: **Filter Rules Support.** Corresponding to Table 2.2, the approach to adblocking affects the type of filter rules that are supported. "Browser" denotes approaches that render web content using browser extensions, custom browsers, and web views. "Cross-app" approaches utilize local VPNs to decrypt the network traffic of a device to apply filter rules and block ads and tracking. "Cross-device" applies the rules on the DNS traffic for all devices within a particular network. ● = fully supports, ◐ = partially supports, ○ = no support.

2.3 Adblocking Beyond the Web

Adblocking is possible beyond the web. Users have several platform choices that provide app stores, such as mobile [121], smart TVs [137], and extended reality devices [132]. As illustrated in Table. 2.3, adblocking can apply within a browser as an extension or integrated directly into the browser, apply across apps, and apply across devices. For example, users can install applications that integrate with native browsers, acting as browser extensions to block ads and tracking (*e.g.,* AdGuard on iOS). In addition, users can install applications that utilize web views (*i.e.,* objects that render web content) for adblocking. However, these solutions confine adblocking to applications that render web content. To achieve adblocking across apps, including native apps, users rely on on-device VPNs that intercept the network traffic of the device to block ads, such as the AntMonitor app for Android [121]. Moreover, adblocking is possible across different devices at the network level. Users can install DNS-based adblockers for home networks, such as Pi-Hole [108] and AdGuard Home [12], which block ads and tracking for devices that connect to the same network. A user-friendly solution involves connecting to DNS servers with adblocking functionality, such as the AdGuard DNS server. However, this assumes that the user trusts the DNS server.

Limitations. These aforementioned approaches still depend on filter rules but with several limitations. Local VPN apps incur efficiency drawbacks, as they must decrypt the network traffic to apply URL-based filter rules. Unlike the web, filter rules that hide HTML elements and abort JS execution cannot be utilized. Even worse, DNS-based adblocking restricts filter rules to the domain or hostname form, *i.e.,* not URL-based rules. Prior work investigated the efficacy of DNS-based adblocking for smart TVs [137] by applying four popular filter lists: the default Pi-Hole list [109], the Firebog list that contains rules for smart TVs [140], the Mother of All Adblocking list curated for mobile devices [95], and a commercial filter list for smart TVs called StopAd [79]. They conclude that these filter lists either miss blocking ads or cause functionality breakage. There is a need for platform-specific filter lists.

Takeaways. Challenges to creating and maintaining filter rules and how they can be applied within browsers, cross-app, and cross-device, motivate our work, as described in Tables 2.1, 2.3. First, adblocking within browsers supports the widest range of filter rule types, causing the arms race between publishers *vs.* adblockers to be more accelerated. In other words, if ads are easily blocked due to the availability of filter rule types, more effort will be put forth to evade their blocking. Chapter 3 examines this accelerated arms race. Second, across all platforms, there are scalability and efficacy challenges. Filter rules need to work for thousands to millions of websites and applications. They must be effective with minimal breakage. This problem is exacerbated when there is a constraint on the rules that can be created and applied (*e.g.,* cross-app and cross-device columns of Table 2.3). Chapter 4 takes a step in addressing these challenges for the web by developing an automated approach to filter rule generation using reinforcement learning.

Chapter 3

CV-Inspector: Automated Detection of Adblock Circumvention

3.1 Introduction

The widespread adoption of adblocking has threatened the advertising-based business model of many online publishers [32]. As discussed in Sec. 2.2.2, in response, publishers have deployed anti-adblockers that detect adblockers and force users to either disable their adblockers or sign up for paid subscriptions [17, 126, 106]. However, anti-adblocking has not proven very successful: adblockers can often hide anti-adblocking popups [72, 149, 100] or users mostly choose to navigate away [107, 32]. Some publishers have resorted to outright circumvention of adblockers. There are now dedicated third-party *circumvention (CV) services* that help publishers re-insert ads by bypassing adblockers. Examples include AdThrive [18], AdDefend [11], and Publica [112]. These CV services are different, and more advanced, than anti-adblockers. While anti-adblockers generally initiate a dialogue with users [123], CV providers try to sneak ads without giving users any notice or choice [116, 115, 80]. More specifically, CV services re-insert ads by evading filter lists [25], such as the community-driven

EasyList (EL) [54], used by adblockers to block ads [47, 103, 42, 19].

The adblocking community has taken notice of the aggressive circumvention tactics used by CV services. Most notably, Adblock Plus (ABP) [10] established a dedicated anti-circumvention (anti-CV) effort that is centered around a new dedicated filter list, the *anti-circumvention list (ACVL)*, to counter these CV services [8, 90, 61]. ACVL supports an extended syntax with advanced capabilities, such as to hide DOM elements based on a combination of CSS styles and text, beyond the simpler rules supported by EL [94]. Concurrently with ABP, other adblockers, such as uBlock Origin [135] and AdGuard [13], also incorporate similar advanced anti-CV filter rules [133, 67, 15, 14]. Similar to other adblocking filter lists [54, 55], anti-CV filter rules are curated manually based on crowdsourced user feedback. However, ACVL is curated primarily by a small set of expert list authors instead of the broader community that supports EL. Thus, a key challenge faced by the ACVL curators is keeping up with the fast paced nature of CV services [52]. Our measurements show that the updates to ACVL are made 8.7 times more frequently as compared to EL. Another challenge is that anti-CV efforts are in the public domain, which gives CV providers the opportunity to monitor anti-CV efforts and adapt their evasive tactics accordingly.

To address these challenges, we introduce CV-Inspector, an automated approach to detect whether a site employs adblock CV services. CV-Inspector includes (i) an automated data collection and differential execution analysis for a list of sites of interest; (ii) an algorithm for prioritizing and expediting ground truth labeling; and (iii) a supervised machine learning classifier using features that capture obfuscation of web requests and HTML DOM by CV services. We evaluate CV-Inspector using two real-world data sets. First, we consider the top-20K sites and show that CV-Inspector is able to accurately detect whether or not a site employs circumvention. In the process, we uncover several new sites (including news publishers, adult sites, and niche lower-ranked sites) that successfully employ third-party CV services. Second, we apply CV-Inspector, with ACVL loaded, on a set of sites that

are continuously monitored by ABP, and find that some of them successfully evade anti-CV filters. More importantly, our results show that CV-Inspector can reduce human labeling efforts by 98%, which is a major step in scaling the effort to combat circumvention. To the best of our knowledge, this work presents the first large-scale systematic analysis of adblock circumvention on the web. It provides tools [82] that can significantly automate circumvention detection and monitoring, thus helping to prioritize the efforts of expert ACVL curators, which is a major bottleneck in this arms race.

The outline of the rest of the paper is the following. Sec. 3.2 provides the background of adblock circumvention and related work. Sec. 3.3 provides a longitudinal characterization of the anti-CV filter list and highlights pain-points and bottlenecks. Sec. 3.4 presents the design and evaluation of the CV-Inspector methodology, including the description of the automated web crawling, the differential analysis, the machine learning classifier, and feature engineering. Sec. 3.5 applies CV-Inspector for two different applications: discovering sites that employ CV services in the wild and monitoring sites that are known to employ circumvention to reduce human labeling efforts. Sec. 3.6 concludes with a discussion of potential impact, limitations, and future directions.

3.2 Background on Adblock Circumvention

As noted in Sec. 2.1.2, adblockers rely on filter lists to detect and counter ads. Rules in these filter lists are manually curated by volunteers based on crowdsourced user feedback [19, 125]. Filter rules, as shown in Table 2.2, can block network requests to fetch ads using hostname or path information. In addition, they can hide HTML elements of ads using class names or IDs. As adblocking has gone mainstream [32], publishers have undertaken various countermeasures that can be divided into three categories: whitelisting (Sec. 2.2.1), anti-adblocking (Sec. 2.2.2), and circumvention (Sec. 3.2.1) as described in the following section.

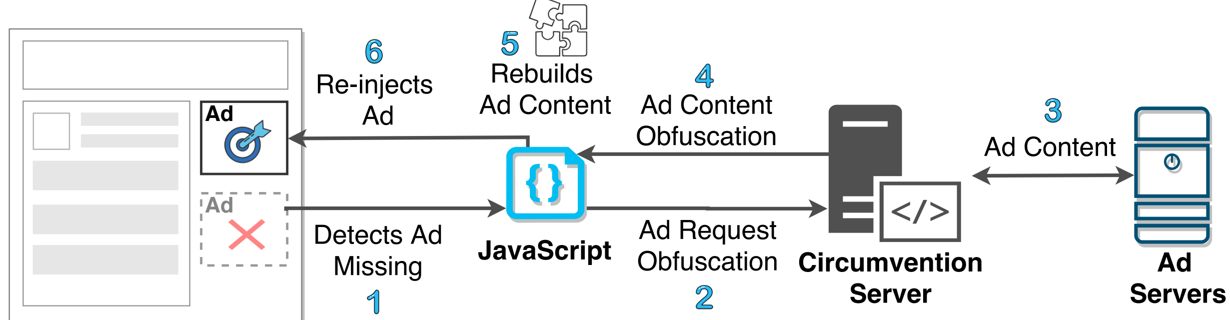


Figure 3.1: **Obfuscation-based Circumvention.** (1) If JS detects that an ad is missing; (2) it sends an obfuscated ad request through a CV server; (3) the server retrieves the new ad from an ad server; (4) the server obfuscates it before sending it back to the browser; (5) JS rebuilds the ad content into DOM elements; and (6) re-injects the ad back onto the page. Compare this workflow with anti-adblocking in Fig. 2.1.

3.2.1 Circumvention

Publishers have recently started to manipulate the delivery of ads on their site to outright circumvent adblockers. Circumvention techniques can be broadly divided into two categories:

Cloaking-based Circumvention. Publishers route ads through channels that adblockers do not have visibility into due to bugs or other limitations. For instance, advertisers used WebSockets to circumvent adblocking extensions in Chrome because of a bug in the WebRequest API that is used by extensions to intercept network requests [26]. More recently, advertisers have used DNS CNAME to disguise HTTP requests to advertising and tracking domains as first-party requests [48, 49]. However, cloaking-based circumvention is not long-lasting because it is neutralized once the bug is fixed. For example, Bashir *et al.* [26] showed that WebSockets-based cloaking was rendered ineffective when Chrome patched the WebRequest bug [110]. Moreover, cloaking is typically not effective against browsers with built-in adblocking because they are not constrained by the extension API used by adblocking extensions. Thus, we do not focus on cloaking-based circumvention in our work.

Obfuscation-based Circumvention. Publishers obfuscate their web content (*e.g.*, URL

path, element ID) to evade filter rules used by adblockers [142]. In contrast to cloaking-based approaches, obfuscation-based circumvention is powerful because it exploits the inherent weaknesses of filter rules — namely, that rules must be precise when targeting what to block (*i.e.*, to avoid false positives) and that they are slow to adapt (*i.e.*, rule updates). Furthermore, obfuscation-based circumvention can allow publishers to seamlessly continue programmatic advertising that is financially more lucrative for publishers than anti-adblocking.

In this work, we focus on obfuscation-based circumvention. Fig. 3.1 illustrates its workflow: (1) JS detects whether an ad is missing; (2) if an ad is found to be missing, then an obfuscated web request is sent to a CV server; (3) the CV server de-obfuscates the request and relays it to the corresponding third-party ad servers or the publisher's ad server to attain the new ad; (4) the CV server obfuscates the ad content and sends it back to the browser; (5) JS rebuilds the ad content into DOM elements; and (6) it re-injects the new ad at a desired location.

Alrizah *et al.* [19] anecdotally showed that EL is ineffective at countering obfuscation-based circumvention. More recently, Chen el at. [42] found that about one-third of advertising and tracking scripts are able to evade adblocking filter rules due to URL and other types of obfuscation. To the best of our knowledge, prior work does not provide large-scale characterization of adblock circumvention or automated circumvention detection in the wild.

Anti-Adblocking vs. Circumvention. Fig. 3.1 compares anti-adblocking and adblock circumvention. Both approaches share the first step, detecting whether an ad is missing. Patently, this is necessary for anti-adblocking. However, for circumvention, it is not a required step but a choice that publishers select to minimize the cost of using CV services.

After the first step, their subsequent steps differ. As shown in Fig. 3.1, different from anti-adblocking, circumvention involves a series of additional steps at the server-side to bypass filter rules and re-inject ads in the client-side browser. Thus, circumvention is a more intricate process. It must deal with the process of attaining new ad content and where to place

them on the page. Recall that it must do this without disrupting the user experience while also evading filter rules. The complexity of circumvention is further denoted by adblockers implementing new advanced filter rules, such as aborting JS execution, to adequately combat it. This is further explored in Sec. 3.3.

As noted before, anti-adblocking and circumvention both aim to affect adblock users only: thus, making differential analysis a suitable technique to detect them. Intuitively, differential analysis endeavors to capture fundamental characteristics of anti-adblocking or circumvention. For instance, with regards to Fig. 2.1, prior work [100, 149] sought to detect the action of step 1 and whether the popup of step 2 was displayed to the user. Note that the outcome of anti-adblocking does not involve ads. On the other hand, our work identifies characteristics of circumvention, described in Fig. 3.1, within actions of steps 2 and 4, and whether ads were displayed as a result of step 6.

However, the differential analysis method proposed in prior work to detect anti-adblockers cannot be directly used to detect adblock circumvention. For example, Zhu *et al.* [149] conducted differential analysis of JS execution to find branch divergences due to anti-adblocking. This technique, if used as is, would incur false positives when a site is able to re-insert ads but *unsuccessfully* displays them due to filter rules hiding the ad element. More specifically, the circumvention approach illustrated in Fig. 3.1 would exhibit a branch divergence at the first step of detecting missing ads, which would be incorrectly considered a positive label (successful circumvention). While CV-Inspector also uses a differential analysis approach that involves loading a page with and without adblocker, it does not aim to capture branch divergences due to anti-adblocking. As we discuss later, CV-Inspector conducts differential analysis of web requests, DOM mutations, and other features to be able to distinguish between *successful* and *unsuccessful* circumvention of adblockers.

3.3 State of Anti-Circumvention

The adblocking community is increasingly wary of circumvention. Most notably, ABP recently started a dedicated filter list, ACVL, to combat circumvention [74]. The filter list is enabled by default in ABP to help block "circumvention ads." This anti-CV list has two key advantages over the standard EL. First, it allows ABP to have full control over filter rule design and management, including pushing the updated rules at a higher frequency (*e.g.*, every hour as opposed to every four days for EL) and without community consensus. Second, it supports advanced filter rules with enriched syntax and capabilities, which are not supported by the standard EL, specifically to counter CV services [65].

3.3.1 Filter Rules Overview

Filter rules can be either simple or advanced. Table 2.2 provides examples and their compatibility with EL and ACVL. We refer to EL types of rules as *simple filter rules*: they can block web requests by matching domains or hide DOM elements by targeting CSS styles or content.

ACVL deploys additional *advanced* rules to combat circumvention: these can abort the execution of JS or hide DOM elements based on computed styles and visibility of content [59]. For example, if "EX" is an JS object that holds circumvention code, then "||a.com#$#abort-on-property-read EX" can block any JS that accesses it. Creating the rule often involves reverse engineering the code to identify that "EX" holds circumvention-related code. Furthermore, a filter rule like "||a.com#$#hide-if-contains-visible-text /Sponsor/" can hide any element containing the visible text "Sponsor." Notably, this differs from simple element hiding because the simple rule only takes into account the existence of text content and not whether it is displayed to the user.

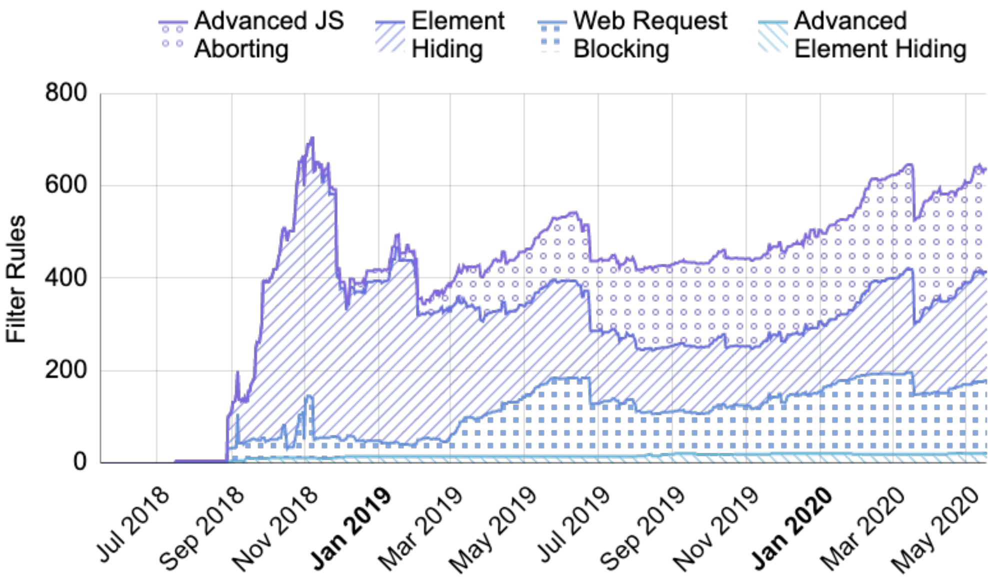


Figure 3.2: **Anti-circumvention List Over Time.** This shows how filter rules from ABP's ACVL have evolved from May 2018 to May 2020 and categorizes them by filter types.

3.3.2 Analysis of the Anti-circumvention List (ACVL)

Evolution of Anti-circumvention Rules. We consider the commit history of ACVL by using its GitHub repository and rebuild the list's filter rules for each day from May 2018 to May 2020 [8]. Fig. 3.2 shows the evolution of the list since its inception in May 2018. The list grew rapidly near the end of September 2018 and peaked at 700 filter rules in November 2018. We see the overwhelming usage of element hiding over other filter types such as web request blocking and advanced element hiding. This can be attributed to the fact that advanced element hiding has a large performance cost (with the use of "window.getComputedStyles"), causing filter list authors to use it sparingly. Also, element hiding may have been more effective in 2018 because JS aborting was not introduced until mid-November of that year [69]. Due to the dependency on element hiding rules, we see that until February 2019, ABP could not prevent the loading of circumvented ads but rather only hide them from the user. Moreover, we see a large drop in element hiding rules (~300 filter rules removed) from November to December 2018. When inspecting the commit changes of that drop, they appear to be

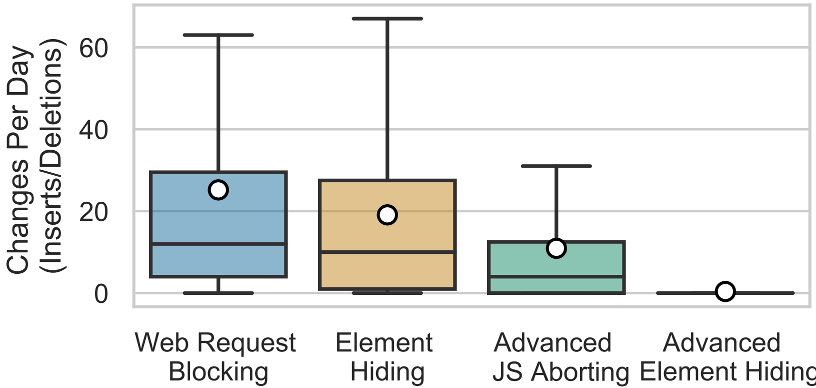


Figure 3.3: **Commits by Filter Type.** A boxplot of commit changes from 2018 to 2020 and categorized by filter types for ACVL. The horizontal lines within the boxes represent the median, while the white circles represent the mean.

cleaning up old filter rules for Czech and German sites [143, 118]. In particular, we find that many element hiding rules are used to target only 13 sites (*e.g.*, novinky.cz and super.cz). This is a downside to element hiding: it must target specific elements resulting in a large number of rules to cover ads even for one site.

Next, we observe that the introduction of JS aborting rules in mid-November 2018 triggers a change in the filter type usage within ACVL. First, the popularity of JS aborting rules denotes its effectiveness against circumvention. Second, it reduces the ACVL's dependency on element hiding because JS aborting prevents ad reinsertion, which results in fewer ad elements to hide. Consequently, this also increases the popularity of web request blocking. This can be due to two factors: (1) once filter list authors understand which JS employs circumvention, they can better find a way to block the script entirely; and (2) CV services rely more on web request obfuscation during that period. Thus, from late 2018 to 2020, we see that the three filter types were used almost equally.

Frequency of Updates. For 2019, which denotes ACVL's first complete year, Fig. 3.2 shows that the number of filter rules has stabilized within the range of 400 to 500 rules. This contrasts with EL's constant growth, which increases at approximately 8K rules per year [125].

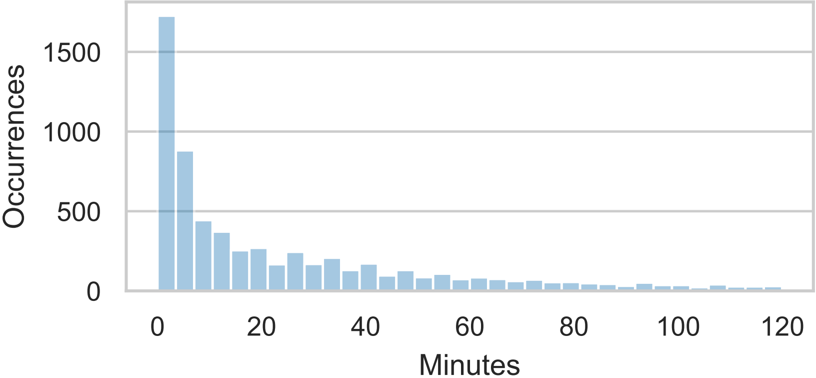


Figure 3.4: **Time between Commits.** The time between commits for ACVL is most frequently within 4 minutes while the average is 2.3 hours.

However, the daily modifications to ACVL remain high. To explore this notion, we review the changes of all commits within a day by using “git diff” and parse each change to categorize them into filter types. Fig. 3.3 reveals the spread of changes per day (defined as the number of inserts and deletions) for each filter type within ACVL. We find that the medians of changes are 12, 10, and 5 for web request blocking, element hiding, and advanced JS aborting, respectively. The median for advanced element hiding remains at zero due to its infrequent changes. Moreover, the frequency of commits persists at a high rate, as indicated by the time between commits, reported in Fig 3.4: it is commonly within 4 minutes. The average time is 2.3 hours, which is about 8.7 times more frequent than EL’s 20 hours [125]. This highlights the accelerated arms race between publishers and adblockers within the circumvention space.

Publishers that Employ Circumvention. We find that commits are generally modifications to existing rules. For web request blocking, curators typically change URL components (*e.g.*, subdomains and paths) or the resource type within the rule. For both simple and advanced element hiding, they often modify class names, IDs, styles, and the DOM structure. For advanced JS aborting, they change the name of the JS object that the rule is targeting.

To evaluate publishers that cause these frequent commits, we identify domains that appear in both insert and deletion lines within a commit. We discovered that the top two sites,

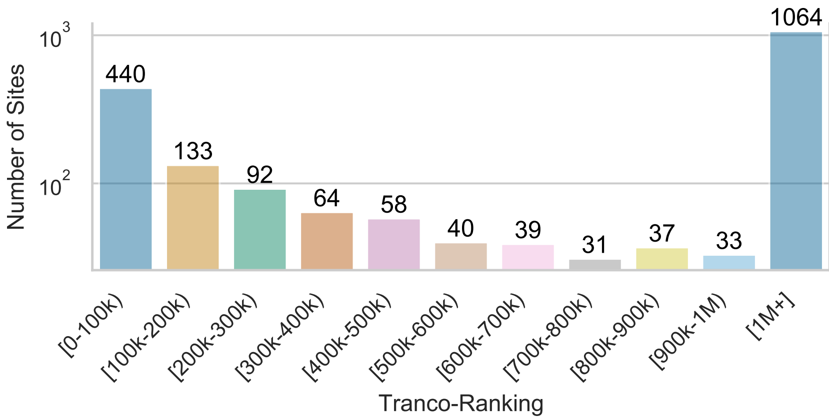


Figure 3.5: **Tranco-ranking of ACVL.** Sites extracted from ACVL and their corresponding Tranco-ranking. We see that there is low coverage of sites for circumvention from ranking 100k to one million. Note that about half of the sites do not appear in the Tranco top one million list (labeled as 1M+).

reuters.com and quoka.de, have triggered 671 changes to rules over a period of 17 months and 269 changes within 18 months. Overall, the top-10 websites that give filter list authors the most trouble have an average ranking of about 10K [111]. However, when considering the top-30, the average ranking is 189K, which can be explained by the fact that many of the sites are from Germany, where most of the ACVL authors reside.

Coverage of ACVL. Next, we investigate the coverage of ACVL on the web, which has not been previously explored. We extract the sites that are specified in the filter rules and map them to their corresponding Tranco-ranking [111] in Fig. 3.5. First, we note that there are about 927 sites that employ circumvention within the Tranco top one million sites, which denotes the low prevalence of circumvention; perhaps, due to the cost of CV providers. Second, we see that ACVL covers about 1064 sites that are beyond the one million (1M) Tranco-ranking, more than twice the amount of coverage when compared to the top-100K.

Furthermore, we see low coverage numbers for the range of rankings between 100K to 1M. We can deduce that the ACVL may lack coverage in two ways. First, advanced rules must specify which sites to target while simple rules can be website agnostic. Second, we previ-

ously saw that ACVL's number of rules has stabilized — showing that ABP is more focused on combating circumvention from a few known sites rather than discovering new sites that employ circumvention. In addition, while EL authors receive help from the community through forums that have up to 23K reports over a span of nine years [19], ACVL authors rely on submitted GitHub issues, with a current total of 379 issues over a span of two years [8]. Thus, significant manual work (*e.g.*, updating rules and discovering new circumvention sites) falls onto the filter list authors.

Takeaways. The number of ACVL filter rules has stabilized in contrast to EL. This can be attributed to two factors: (1) ABP's focus on a few known CV providers; and (2) changes within ACVL primarily being modifications to existing filter rules. Thus, the coverage of ACVL is limited due to the focus on modifying rules rather than discovering new circumvention sites. Moreover, the effort to combat circumvention requires significant effort from filter list authors. ACVL has only 14 contributors with three main contributors: wizmak, arsykan, and Milene [9], who commit five, four, and three times on average per day, respectively. These few filter list authors must undertake a huge effort in keeping rules up-to-date. This motivates our methodology in the next section, which aims at assisting and prioritizing this effort. It provides ways to detect successful circumvention in the wild, monitor the changes in publishers, and be notified when a site has successfully circumvented the adblocker.

3.4 CV-Inspector: Design and Implementation

In this section, we present CV-Inspector for detecting whether a site employs circumvention. Fig. 3.6 presents an overview of our methodology. In Sec. 3.4.1, we present our instrumentation and automated data collection. In Sec. 3.4.2, we apply differential analysis to identify data that is indicative of circumvention. Then, in Sec. 3.4.3 to 3.4.6, we extract features, train, and evaluate our CV-Inspector classifier.

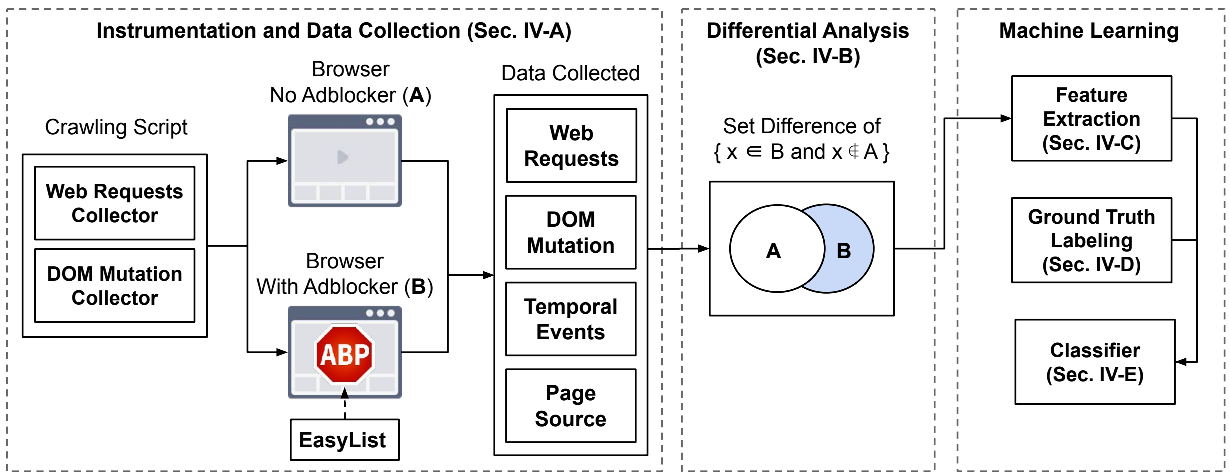


Figure 3.6: **CV-Inspector Workflow.** Given a list of URLs, our crawling script will visit each site four times for: (A) "No Adblocker" and (B) "With Adblocker." With each visit, we collect web requests, DOM mutation events, temporal events (*e.g.*, timestamps and blocked events by the adblocker), and the page source. We take the set difference between the data collected in the two cases, (B)-(A), as websites commonly employ circumvention when an adblocker is on. We use the data to extract most features, train and evaluate our classifier.

3.4.1 Instrumentation and Data Collection

3.4.1.1 How we collect data

Our crawling script takes as input a list of websites for which we collect data. For each page load of a site, we wait for 25 seconds: we denote this as a "page visit." Page load times are commonly less than a minute as they affect the search ranking of sites. As shown in Fig. 3.6, we visit each site for a total of eight times. As a result, we select 25 seconds to not significantly slow down CV-Inspector, which is inline with prior work [149].

No Adblocker vs. With Adblocker. Since websites typically employ circumvention only when an adblocker is present, we utilize differential analysis to obtain insights into circumvention "signals." For each website, we collect data for two different cases: (A) "No Adblocker" and (B) "With Adblocker." For "No Adblocker," we load each site four times and take the union of the collected data in order to capture the dynamic nature of a site because it can re-

trieve ads from different ad servers. This is a heuristic but justified choice: we experimented with loading the same page multiple times and found that the number of contacted domains plateaus at four. We will refer to these "four page visits" per case, throughout the paper. We repeat the same process for "With Adblocker." In addition, we use ABP and configure it to use EL. We deselect the "Allow Acceptable Ads" option as we want to make sure ads are shown due to circumvention and not because it was whitelisted. Furthermore, this gives sites the best chance to circumvent the adblocker and the best opportunity for us to capture it.

Landing Page and Sub-pages. Our crawling considers both landing pages and sub-pages. This is critical because sites may not employ circumvention in their landing pages but rather wait until the user clicks into a sub-page to show circumvented ads (*e.g.*, maxpark.com). To find a sub-page, we inject JS into the landing page to retrieve all URLs from hyperlink tags. We select the first-party link with the longest number of path segments. We use the intuition that the deeper the user explores the site, the more interested the user is in the content, thus increasing the chance that the site would serve ads. We find that this methodology works well for sites that have articles. To further ensure that we find a sub-page with ads, we ignore informational pages using keywords (*e.g.*, "contact," "login") within the path. To only consider pages with content, we further ignore first-party links that have extensions (*e.g.*, ".tar.gz," ".exe"), to prevent downloading external files.

Automatic Collection. We use Selenium [101], a framework to automate the testing of websites, to implement the crawling process. We select Chrome (version 78) [45] as the browser due to its popularity. As depicted in Fig. 3.6, we create two Chrome profiles. One profile for the "No Adblocker" case, where we include the web request extension and DOM mutation extension. The second profile is for the "With Adblocker" case, where we also include the custom ABP extension that only loads EL. In order to have a consistent behavior with ABP, we only use one version of EL and the ACVL from March 13, 2020. Then, we configure Selenium to disable caching and clear cookies to have a stateless crawl. For scalability

purposes, we utilize Amazon's Elastic Compute Cloud (EC2) and select the "m5.2xlarge" instance that allows CPU usage without throttling [20]. We create a snapshot out of the setup using Amazon Machine Image (AMI) [21], which allows us to spawn many instances of EC2 for data collection.

3.4.1.2 What data we collect for each page

Next, we describe the types of information we collect for each site. We are interested in how the site changes from "No Adblocker" to "With Adblocker," at four vantage points:

1. Web requests: HTTP incoming and outgoing requests.
2. DOM mutation for nodes, attributes, and text.
3. Time stamps of all events like web requests, DOM mutations, and blocked events caused by ABP (*i.e.*, when a filter rule is matched, see Table 2.2).
4. Page source code of the site (*e.g.*, HTML, text, inline CSS, and inline JS).

We also collect screenshots, which are capped at 1925x3000 to deal with websites that can infinitely scroll. Screenshots are useful as we use them to verify our ground truth in Sec. 3.4.4. Next, we explain how this collected data can reveal obfuscation-based circumvention employed by the site.

1. Collecting Web Requests. Circumvention providers often randomize subdomains and paths as an obfuscation technique to retrieve new ad content for reinsertion, going beyond simply rotating domains [72, 19], as illustrated in Fig. 3.1. Capturing web requests can help identify this behavior. Examples are provided in Sec. 3.4.3. We implement a Chrome extension to collect web requests by hooking into the Chrome Web Request API [64]. This API streamlines the flow of web requests into various life-cycle stages that developers can

easily subscribe to. Specifically, we hook into "onSendHeaders" to collect outgoing HTTP request headers and "onCompleted" to collect incoming HTTP response headers of successful requests. To collect web requests blocked by ABP, we hook into "onErrorOccurred" and look for status code "ERR_BLOCKED_BY_CLIENT."

2. Collecting DOM Mutation. Fig. 3.1 shows that re-injected ads are often reconstructed in step 5 and may not have the same DOM structure as the originally blocked ads. Capturing how the DOM changes as the page loads can help uncover these particular actions. We build a Chrome extension that uses DOM Mutation Observers [98] to collect DOM changes. The extension compiles events such as new nodes added (*e.g.*, an ad image being added), nodes removed (*e.g.*, a script being removed), attribute changes (*e.g.*, an ad element from height 0 to 280px), and text changes (*e.g.*, anti-adblocker popup text). Furthermore, recall from Table 2.2 that an adblocker can do element hiding. We capture this by instrumenting the ABP extension (version 3.7) and hook into methods that hide elements when a filter rule is matched to label the elements with a custom HTML attribute "abp-blocked-element," shown in Listing 3.1. Since this causes a DOM attribute change, we consider this as part of the DOM Mutation information.

3. Collecting Temporal Information. Since circumvention is typically a reaction to ads being blocked, timestamps of changes on the page can reveal how adblockers and circumvention code interact with each other. Thus, we record and consider timestamps for web requests, DOM mutation, and blocked events. For completeness, when we consider the ACVL in Sec. 3.5.2, we hook into methods that abort the execution of JS to capture JS blocked events as well.

4. Collecting Page Source with Annotations. We use Selenium to save the page source of the site at the end of the page load time. It gives us information about the state of the site such as the HTML and text, inline CSS, and inline JS. In addition, it contains the

Listing 3.1: **Page Source Annotations.** Highlighted in blue, attribute "abp-blocked-element" denotes that the adblocker has blocked the element. While attribute "anticv-hidden" means that the img is not visible (not related to the adblocker). All visible images and iframes are labeled with their offsetwidth and offsetheight to give a more accurate representation of the page.

```
0
1  <div abp-blocked-element="true">
2      <img src="https://ads.com/banner.jpg"/>
3  </div>
4  <div class="mobile">
5      <img src="https://b.com/logo.jpg"
6      anticv-hidden="true"/>
7  </div>
8  <iframe src="https://b.com/ad" height="90"
9      anticv-offsetwidth="728"
10     anticv-offsetheight="90">
11 </iframe>
```

annotated elements that are hidden by the adblocker, as shown in Listing 3.1. Furthermore, since the page source does not provide the actual visibility state of images and iframes, we inject JS to annotate these elements with a custom attribute "anticv-hidden" detailed in Listing 3.1. We extract all images and iframes and consider the following cases. First, if the element's "offsetParent" is null and its "offsetWidth" and "offsetHeight" are zero: this denotes that the element is hidden due to its parent being hidden. Second, otherwise, we use "window.getComputedStyles," which provides us with the final styles that are applied to the element. We consider styles such as "display: none" and "opacity <= 0.1" to see if the element is hidden. Third, we treat elements with a width and height of less than or equal to two as hidden. This filters out pixel elements used for tracking. We further use these annotations for feature extraction, as described in Sec 3.4.3.

3.4.1.3 Tools and Limitations

Using Amazon's EC2 and AMI, our methodology is scalable (*e.g.*, multiple instances can be initiated to fit the problem) and configurable (*e.g.*, number of sub-pages to find, which

filter list to load). However, it also has its limitations. First, some sites utilize Cloudflare's protection against web crawlers using captchas, which prohibits CV-Inspector from accessing the page. Second, Selenium may not properly produce screenshots, which depends on how body styles are applied. We address this limitation by first checking whether the height of the body is zero. If so, then we check the next immediate child element of the body to see if it has a height to capture, and so on. Third, when discovering sub-pages, we do not consider links from non-hyperlink tags or if the site is utilizing JS to redirect users upon a click. Finally, recall that we wait for 25 seconds during each page visit, which might miss some behavior on sites that need longer to load. This is a parameter to tune: longer crawling times are possible at the expense of slowing down CV-Inspector.

3.4.1.4 Datasets

We apply our methodology and collect datasets, summarized in Table 3.1, which we then use for different purposes throughout the paper. For each of these datasets, we start from a list of URLs, apply the methodology described earlier in this section, and we collect the four types of information, referred to as "collected data" in Fig. 3.6: web requests, DOM changes, temporal information, and page source with annotations. The top three datasets in Table 3.1 are collected using our methodology based on a given list of sites: ACVL sites, Tranco's most popular sites and Adblock Plus Monitoring. The first two are publicly available.

ACVL has been extensively discussed in Sec. 3.3 and includes sites that currently employ, or had employed in the past, CV services; we use this list to find positive samples. We use Tranco ranked sites in two ways. First, since circumvention is hard to find, we use the Tranco top-2K sites within our ground truth dataset (GT) to ensure that it covers popular sites. Second, we use the Tranco-20K dataset (which excludes the top-2K) to test our classifier on popular sites that matter to users. The third dataset, internally maintained by ABP, contains sites that employed circumvention at some point and ABP continuously monitors

Dataset Name	List of Sites Crawled	# Pages & Sub-pages
	Obtained by crawling a given list of sites	
ACVL sites	Sites extracted from ACVL (public [8])	3K
Tranco	Most popular sites (top-20K) at tranco-list.eu (public [111])	32K
Adblock Plus Monitoring	Sites that ABP monitors (maintained and provided by ABP)	360
	Derived from ACVL & Tranco, used for ML training & testing	
Candidate for labeling (CL)	ACVL ∪ Tranco top-2K	6.2K
Ground Truth (GT)	Subset of sites from CL that are inspected and labeled (positive or negative) for circumvention	2.3K
Tranco-20K	Tranco top 2K-20K (excluding the top-2K used in CL)	29.3K
Ground Truth Positives (GTP)	Subset of GT with only positive labels	700

Table 3.1: Dataset summaries and terminology used throughout the paper. Each of the original datasets is obtained by crawling the corresponding list of sites (and sub-page) and collecting all 4 types of data (web requests, DOM changes, temporal, and page source).

them to see if ACVL is still effective on them. We refer to sites that are closely monitored by adblockers as "sites of interest." Generally, this means that the sites affect a large portion of adblock users (*i.e.*, in terms of popularity) or that the sites have caused users to submit feedback about them.

The bottom part of Table 3.1 summarizes our three original crawled datasets that we use for training and evaluating our classifiers in Sections 3.4.4, 3.4.5, and 3.5.

3.4.2 Differential Analysis

3.4.2.1 Set Difference

Our intuition is that the behavior observed when an adblocker is used ("With Adblocker") is different from the behavior observed when there is no adblocker ("No Adblocker"). This is

likely due to CV services being triggered. Recall from "No Adblocker vs. With Adblocker" of Sec.3.4.1.1, that we need to account for the dynamic nature of websites. Therefore, first, we take the union of the datasets collected across all four page visits in each case. Then, we take the difference of the two union sets ("With Adblocker" minus "No Adblocker"). Next, we elaborate on what differences we examine for each of the four types of data collected.

First, for web requests, circumvention services can serve content behind first-party domains. Therefore, we cannot simply do a set difference on the domain level for web requests, which would eliminate the presence of the circumvented ads. Instead, we do a set difference based on the fully qualified domain and its path while disregarding the query parameters. Second, for DOM mutations, we create a signature for each event based on the element's attribute names, tag name, parent tag name, and sibling count. We do not depend on the value of attributes because they can be randomized [19], which would introduce more unrelated events to circumvention. Instead, we rely on the length of the value within our event signature. For a simple example, if the element is "<div class='ererke434'>," we would consider it as "div_class9." Third, for temporal information, we first extract features per visit then average them within their respective cases, then we apply the set difference. Fourth, for page source, we do a set difference based on words for text differences. For example, a text change event with an old value of "Please subscribe to our content" and a new value of "Please disable your adblocker to view our content," will result in a set difference of "subscribe, disable, your, adblocker, view."

3.4.2.2 Cleaning the Data

Recall that we load each site four times to capture its dynamic content. A side effect is that we end up with data (*e.g.*, web requests and DOM mutations) that is not necessarily related to circumvention, and can be due to tracking, discernible non-ad resources, dynamic content, etc. We filter these out before extracting features for circumvention. First, for web requests,

Web Request Features	**Top**
Number of content-types	✓
Entropy of subdomains, paths, query parameters (by content-types and first/third-party)	✓
Number of Mismatches of URL extension and content-type	
Number of Mismatches of loaded resources	

DOM Mutation Features	**Top**
Number of DOM attribute changes (display, class, etc)	✓
Number of DOM nodes removed (iframes, etc)	✓
Number of elements blocked by EL (imgs, iframes, etc)	✓
Number of DOM nodes added (a, imgs, etc)	

Temporal Features	**Top**
Number of blocked events (in first 12sec of page visit)	✓
Number of blocked events (in second 12sec of page visit)	
Average cluster size of DOM mutations over time	

Page Source Features	**Top**
Number of iframes and images in ad positions	✓
Number of distinct words, characters, and newlines	
Entropy of subdomains, paths, query parameters of visible iframes and images contained in hyperlinks (with target or rel attributes)	

Table 3.2: There were 93 features in total in these 4 categories for CV-Inspector. Those marked as "Top" were in the top-10 most important features in Sec. 3.4.5.

we identify tracking, social, and anti-adblocking requests by applying EasyPrivacy [55], Adblock Warning Removal List [5], Disconnect.Me [51], and uBlock Origin's GetAdmiral [134] filter lists. To filter out the requests, we use Brave's Adblock engine [34], a filter list parser that supports EL-compatible rules. Second, we keep third-party ad resources by looking at ones that have content-length larger than 2 KB and have a max-age (within cache-control headers) shorter than 40 days. We conclude on these numbers by inspecting resources that were blocked by ABP. This gives us a profile about what content-length and max-age ad resources should have. Third, we only consider successful web requests (*e.g.*, HTTP status code 200) and discard the ones that involve redirection, errors, or no content (*e.g.*, HTTP status codes 304, 400, 204). This is because circumvention related web requests should have content such as JSON (that may define ad content) and JS (code to re-inject ads).

Listing 3.2: **Obfuscated URL Example.** Taken from psychologyjunkie.com, we compare a normal URL with an obfuscated one where subdomains & paths are randomized. Although truncated, the path can reach up to 6K in length. The entropy of the subdomains for the regular and obfuscated URLs are 1.58 and 2.25, respectively. Their first path segments would have entropy of 1.79 and 4.56. As expected, the obfuscated strings have higher entropy.

```
0
1 /* Regular URL */
2 https://cdn.convertkit.com/assets/CKJS4.js
3 /* Obfuscated URL */
4 https://h239rh.lmyiwaakn.com/q08HqaNP1NUGrt
5 d4qtgA1agJ2JAHpqoDo9QDqqYApt14qaoF1dZ0...
```

3.4.3 Feature Extraction

Next, we describe the features that we extract from the cleaned set difference to capture circumvention. Not all features involve set differences, *e.g.*, blocked events only appear in the "With Adblocker" case. Table 3.2 lists the features that we explored and highlights those that ended up being the top-10 most important features. Then, we evaluate those features and explain our intuition of why they can capture the presence of CV services.

1. Web Request Features. One widely used obfuscation technique is to randomize URL components and other features extracted from web requests, resulting in noticeable differences between "No Adblocker" and "With Adblocker" cases. Listing 3.2 shows a comparison between a regular URL and an obfuscated one by circumvention. To capture this randomization, we treat URL components, such as subdomains and paths, as strings, and we calculate their Shannon entropy, based on the frequency of each character occurring in the string. The idea is that randomized strings will have higher entropy. An illustrative example is shown in Listing 3.2. As expected, the obfuscated strings have higher entropy for both subdomains and paths. We further split web requests up into first-party and third-party sets. In addition, we count the number of different content-types extracted from their response headers. Also, we look at mismatch cases, such as when a web request ends with a ".jpg" extension but its content-type is "application/javascript." We look at whether a particular path loads

different amounts of resources. For instance, when a path “a.com/images/” loads 10 images with the “No Adblocker” case but then loads 15 images for the “With Adblocker” case.

2. DOM Mutation Features. DOM mutation features can uncover behavior such as when new ad-related elements are added. For nodes being added and removed, we focus on element types that can be associated with ads such as “<a>,” “<imgs>,” and “<iframes>.” For attribute changes, we focus on changes such as the class attribute, visibility styles like display and position, and the height of the element. Moreover, we count the number of DOM attribute changes that involve “abp-blocked-element,” which denotes the number of elements blocked by EL.

3. Temporal Features. We expect that a site would exhibit different behavior (events) over time when employing circumvention, as depicted in Fig. 3.1. Therefore, we examine the timing of events to extract temporal features. Fig. 3.7 details how we capture differences in DOM mutations over time by utilizing spikes, clusters, and cluster sizes. By considering the cluster size, we can identify bursts of DOM mutations and how prolonged they are. For “With Adblocker,” we see fewer DOM mutations within the first five seconds, perhaps due to many blocked events in the beginning. However, after that, we see more bursts of DOM activity; notably, within the 12–18 seconds that are not present in the “No Adblocker” case. This is captured by the smaller average cluster size. Interestingly, this turned out not to be a top feature. We deduce that this is because not all circumvention techniques cause large DOM mutation changes. For instance, a site can load in a static ad and use a simple ad structure, as shown in Listing 3.3. We further discuss circumvention techniques in Sec. 3.5.1.2 and Table 3.5.

Since blocked events (*i.e.*, any matching of filter rules in Table 2.2) can happen for sites that do not employ circumvention, we want to investigate whether the timing of blocked events can signal circumvention. Recall that we visit each page for 25 seconds, a parameter value chosen for reasons explained in Sec. 3.4.1.1. We compute the number of blocked events in

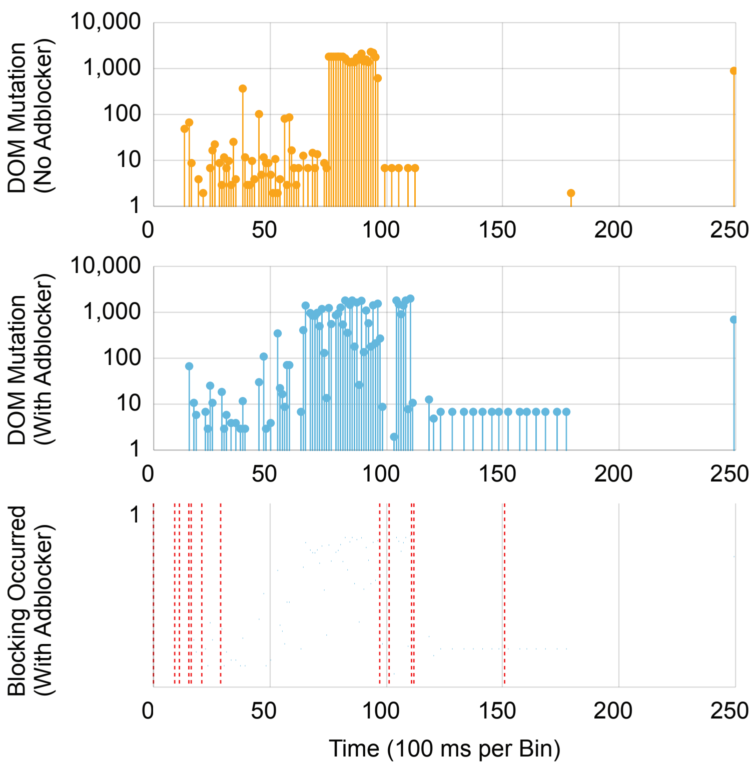


Figure 3.7: **Example of Temporal Features.** We show the number of DOM mutations (spikes) over time for "No Adblocker" and "With Adblocker" (with the corresponding blocked events). We define a cluster of activity as consecutive spikes (no more than one bin apart) and the cluster size as the number of bins that it spans. The top figure shows the "No Adblocker" case, which has 9 clusters with an average cluster size of 8.33. In the middle figure, we show the "With Adblocker" case, which has 22 clusters with an average size of 3.86. In the bottom figure, the dashed vertical lines represent whether blocking events occurred. The majority of blocking happened within the first 12 seconds when compared to the remaining time (*e.g.*, 11 events *vs.* 1 event).

the first or second 12 seconds of the page visit. We initially thought the second half would be a differentiating feature, as the page would exhibit the action of re-injecting ads and the adblocker would then once again block those ads. However, we observed that the first half was more important, as shown in Fig. 3.7. This may be because loading ads is a priority, leading to the blocked events happening at the beginning of the page load. Also, filter rules often aim to stop circumvention at the earliest possible point. Ultimately, adblockers are more aggressive against sites with circumvention, and therefore, cause more blocked events.

4. Page Source Features. Page source features characterize the state of the site at the

Listing 3.3: **Simple Ad Structure.** An example of a simple ad structure that can be used during ad re-insertion instead of an iframe.

```
0
1 <a href="https://www.512xiaojin.com"
2      target="_blank" rel="nofollow">
3      <img src="https://1.bp.blogspot.com/
4      -YTj2YjT6ODw.gif" />
5 </a>
```

end of our page visit time. These features convey whether circumvention was successful by identifying possible ads that are still visible on the page. We discover that circumvention exhibits behavior such as altering the DOM structure of the ad to circumvent adblockers, while re-injecting the ads back to specific, and often the same, locations.

First, we target specific DOM structures that hold ads such as images or iframes. For images, we select those that are contained by hyperlink elements ("<a>") with attributes "target" and "rel," as shown in Listing 3.3. The "target" attribute defines how the browser behaves after a user clicks on the link such as opening up in a new window or tab. The "rel" attribute defines the relationship between the current page and the outgoing link. We can use this to infer that if the outgoing link is also third-party, then it is likely to be an ad.

Second, we identify possible ad locations that can be utilized for re-injection. We use the "No Adblocker" page source and extract all iframes. We then dynamically create CSS selectors for the iframes, specifying at least three levels of ancestors to make sure the selector is specific enough. We then use these selectors on the page source of the "With Adblocker" side and count the number of images or iframes that remain. To deal with sites that randomly alter their element attributes, we do a second search (when the first search does not match any elements) with more generic selectors by looking at the existence of attributes and not the values of them. For instance, a selector of "div > div[opacity='1'] > div[class='rerejhf']" will turn into "div > div[opacity] > div[class]."

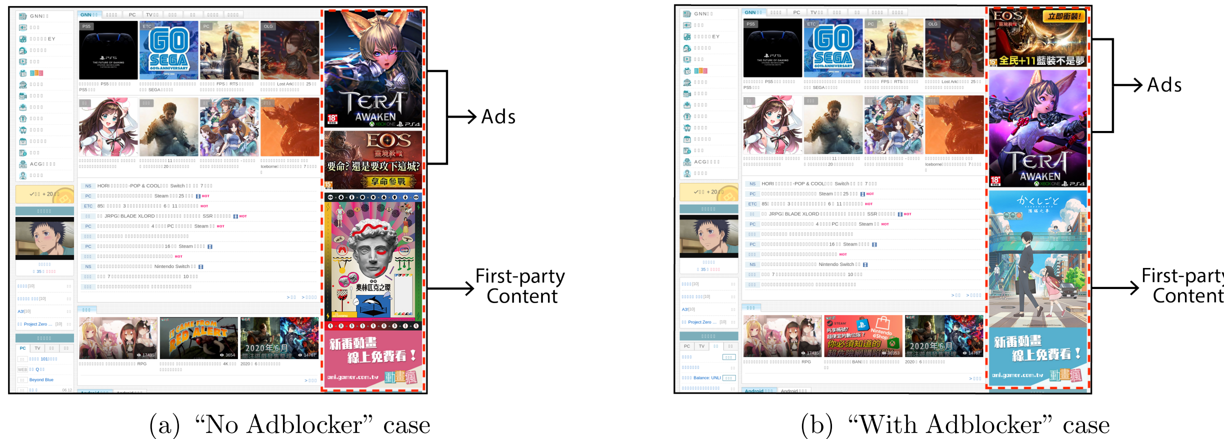
For both of these cases, we make sure that iframes and images are visible and not hidden by the adblocker or pixel-size used for tracking. This is possible by using our annotations from Listing 3.1 to ignore elements that are invisible to the user.

3.4.4 Ground Truth Labeling

Let us revisit Sec. 3.4.1.4 and discuss how we use the original datasets, shown in the top two rows of Table 3.1, to create our GT dataset, for training our classifier.

Why Positive Labels are Important. A major challenge for our GT dataset is that positive samples (*i.e.*, sites that successfully employ circumvention) are rare and hard to find. First, there are simply not many sites that employ circumvention today. For example, in Fig. 3.5, only 927, out of the top one million Tranco sites, utilize circumvention. Second, we define positive labels as not only attempting circumvention, but also successfully circumventing adblockers, which further reduces their number. Conversely, negative labels are easy to discover because they correspond to sites that do not attempt circumvention or to sites that were unsuccessful at evading the adblocker. For instance, see the imbalance in Table 3.4. Furthermore, human inspection and labeling of sites is a labor-intensive process. To resolve these challenges, we devise a methodology that reduces human labeling efforts while finding many positive labels.

Candidates for Labeling (CL). We start from a list of URLs that we consider candidates for labeling: this includes 2K domains extracted from the ACVL, as described in Sec. 3.3.2, and popular Tranco top-2K sites. Domains extracted from ACVL are not guaranteed to have positives, because compatible rules from ACVL can be transferred to EL, thus EL can deal with circumvention for some sites. Furthermore, since Fig. 3.5 reveals that many ACVL domains are beyond the one million ranking, we also consider the Tranco top-2K sites as candidates for labeling, to include more popular sites of interest. We then crawl the sites



(a) “No Adblocker” case (b) “With Adblocker” case

Figure 3.8: **Example of “Suspicious Content.”** The website, *gamer.com.tw*, shows suspicious content on the right sidebar outlined in red. Note that the three small images change between the (a) “No Adblocker” and (b) “With Adblocker” sub-figures. Although the content may look like ads, it could also be benign content related to gaming. Using a browser, we looked at their outgoing URLs and observed that the two smaller images for Tera Awaken and EOS are ads, while the third image links to a first-party page. Since there are still ads displayed in (b) “With Adblocker,” we label this example as a positive label.

using our data collection methodology, depicted in Fig. 3.6, and end up with approximately 6.2k sites (including sub-pages) for our CL dataset.

Labeling Each Site. We label each site, in our CL dataset, as either successful circumvention (positive label) or not (negative label). We capture a screenshot each time we visit a page and depend on them to label our sites. Our labeling methodology is as follows. First, we open up screenshots from “No Adblocker” case and identify where ads are shown. Then we open up screenshots from “With Adblocker” and compare them to see if the ads are removed. If an ad is still visible, we label the site as positive; otherwise, we label it as negative. Second, there may be “suspicious content.” For instance, ads can look similar to page content rather than common ads, either because they lack transparency (*e.g.*, not annotated by “Advertisement” or “Sponsored”), or they may be closely related to the site content. Fig. 3.8 illustrates an example of such “suspicious content”: gaming ads are displayed for a gaming site, “gamer.com.tw,” which makes it difficult to tell whether they are ads or first-party content. To settle these cases, we visit the site on our Chrome browser and

set up ABP with the same configuration (settings and filter lists) as our data collection. This allows us to further verify whether the content was an ad by looking at the outgoing link or testing it out by clicking on it. If the content is indeed an ad that goes to a third-party site, we label it as positive. In our GT dataset, we encountered "suspicious content" only 69 out of 2321 times, thus making it a corner case.

As described, our labeling methodology relies on using screenshots. Recall from Section 3.4.1.1 that for a given site, we visit it four times for the "No Adblocker" and "With Adblocker" cases, which corresponds to four screenshots for each case. An alternative approach to labeling would be to use a browser to check the site, which can produce higher quality labels. For instance, the browser allows us to view the entire site as opposed to the limited height of the screenshots, which is capped at 3000px to deal with infinitely scrolling sites. However, the browser approach increases human labeling efforts. Screenshots offer an attractive compromise: they allow us to quickly compare the four page visits of "No Adblocker" and "With Adblocker" with each other, without setting up our browser and loading the sites four times per case.

Prioritizing Which Sites to Label. Labeling is time-consuming and is a well-known bottleneck in all communities that maintain filter lists, including EL and ACVL. We develop a heuristic for prioritizing which sites from CL to inspect and label first to rapidly discover positive labels and minimize the overall effort. We employ an iterative process shown in Fig 3.9.

Bootstrapping. We start from CL and perform outlier detection using Isolation Forest [119]; our intuition is that sites that utilize circumvention are drastically different from those that do not. However, not all outliers have circumvention, as there can be other reasons why a site behaves differently, such as displaying more page content when ads are not displayed. Therefore, we still need to inspect and label these initial (108) outliers, and we find 56 positive labels. Next, we order the remaining sites extracted from ACVL by Tranco ranking, and pick the top-400 sites. Our intuition comes from Fig. 3.5, where there are around 400

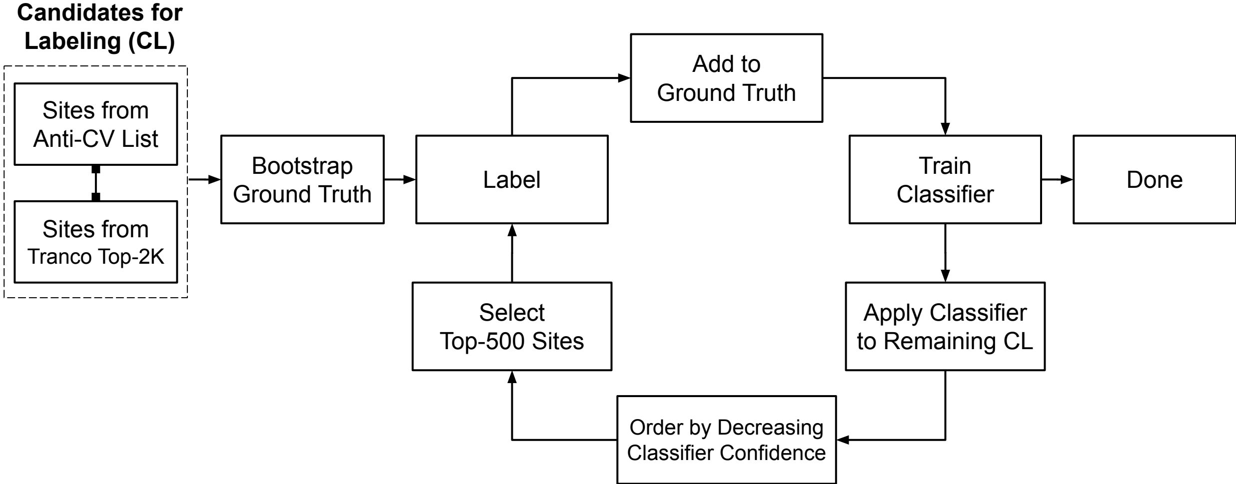


Figure 3.9: **Labeling Methodology:** We start with a list of sites from both ACVL and Tranco top-2K, as Candidates for Labeling (CL). We develop an iterative process for prioritizing which (500 in a batch) sites to inspect and label next, then add them to ground truth. We bootstrap a classifier by using outlier detection to find positive labels. In each iteration, we apply the classifier on the remaining sites in CL, sort the sites by decreasing classifier confidence, and inspect and label the 500 sites where the classifier is most confident. Compared to picking randomly 500 sites to label, this heuristic prioritization discovers more positive labels. For example, see Fig. 3.10 between "Iteration Zero" and "Iteration Zero Random." We add the newly labeled samples into our ground truth, retrain our classifier, repeat the process for two more iterations, and declare "Done" when the performance converges, as shown in Fig. 3.10. We combine all labeled data into our Ground Truth (GT) dataset.

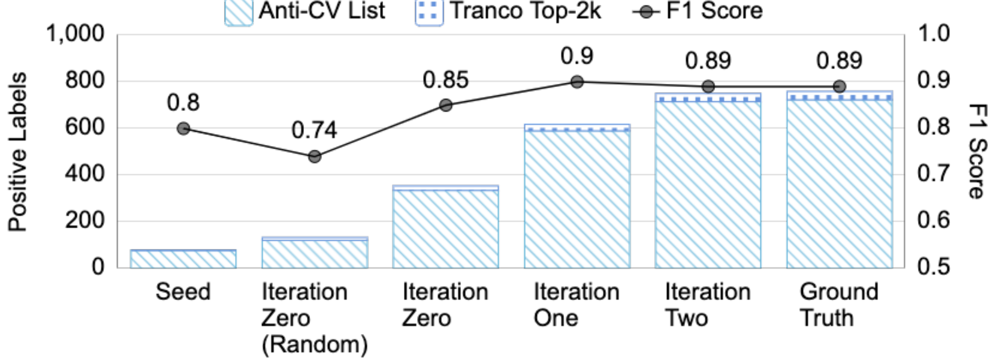


Figure 3.10: **Positive Labels and F1 (per Iteration):** For our ground truth, we show how many positive labels (sites with successful circumvention) were discovered within each iteration. When we compare iteration zero and the randomly chosen iteration zero, we find that our methodology discovers twice as many positive labels. We see that by the end of iteration two, we receive diminishing returns on our classifier performance based on its F1-score. Note that we only find 55 positive labels from the Tranco top-2K overall.

sites in the Tranco top-100k sites. We balance our ground truth with the most popular sites in the ACVL so that our classifier can generalize well in the wild. We merge the labeled outliers with the top-400 sites in the ACVL to obtain our first batch of ground truth with $\sim$500 sites. We train our classifiers on this GT.

Iteratively enhancing the ground truth. We apply the classifier on the remaining sites of CL, sort the sites by decreasing classifier confidence, and inspect and label the 500 sites where the classifier is most confident. We add the newly labeled samples into our ground truth, retrain our classifier, and repeat the process. In each iteration, we choose and label 500 sites and add them to the ground truth, until the performance converges. Fig. 3.10 shows diminishing returns after iteration 1, thus we stop at 2 iterations. The main advantage of prioritizing which sites to label is that it discovers more positive labels in each iteration, compared to *e.g.* choosing 500 random sites to label. This saves human effort, which is the main bottleneck. Fig. 3.10 compares Iteration Zero (with our choice of 500 sites in decreasing confidence) vs. Iteration Zero (Random choice of 500 sites) and shows that we discover more than twice the positive labels, and we achieve a higher F1.

Ground Truth Dataset (GT). We combine all labeled data (from all iterations, including the randomly selected Iteration Zero) into one dataset, which we refer to as GT. It contains 755 positive labels and 1566 negative labels.

3.4.5 The CV-Inspector Classifier

Training the Classifier. We train a classifier that can detect successful circumvention, using all 93 features extracted in Sec. 3.4.3, and the ground truth obtained in Sec. 3.4.4. We considered different classifiers and observed that Random Forest performs best. We split the GT data into 70/30 for training and testings, respectively, and we perform 5-fold cross-validation. We consider our contribution to lie not in the ML technique itself but in

Label	Precision	Recall	Accuracy	F1-score
CV	0.94	0.84	0.93	0.89
No CV	0.92	0.97	0.93	0.94

Table 3.3: **CV-Inspector Cross-validation Results.** Using a Random Forest classifier, 93 features, and 5-fold validation. The label "CV" means successful circumvention and "No CV" means that sites have no CV activity or failed at CV.

the domain-knowledge that guided the design of differential analysis, feature selection, and ground truth labeling.

Cross-Validation Results. We display the results in Table 3.3. Detecting positive labels (*i.e.* sites succeeding in circumventing adblockers) is of interest for filter list authors such as ABP. Here, we achieve an F1-score of 0.89 and a precision of 0.94. Detecting negatives labels is also important because authors want to be confident when disregarding sites without circumvention accurately: we see an F1-score of 0.94 and a precision of 0.92; this becomes invaluable in the monitoring approach in Sec. 3.5.2 as it reduces human effort.

Important Features. Not all 93 features from Sec. 3.4.3 are equally important. In Table 3.2, we highlight the top-10 most important features. Fig. 3.11 also shows the empirical CDFs (ECDF) of four top-features and illustrates that they can discriminate between sites that employ successful circumvention or not. For example, consider the circumvention technique that randomizes the JS first-party path. We see that the path has much more randomness than sites that did not circumvent the adblocker; see the example in Listing 3.2. Specifically, 40% of sites with circumvention have path entropy of two or less, while it is more than 80% of sites with no circumvention. This captures the fact that publishers can use first-party resources that contains circumvention code to initialize the circumvention process. Thus, randomizing the path can make it difficult for the adblocker to block it. The corresponding ECDF is the most discriminatory, compared to ECDFs of other features, uncovering the fact that randomizing the path is a more effective technique against adblockers. Fortunately, our usage of entropy as a feature captures this difference and can detect the presence of circumvention.

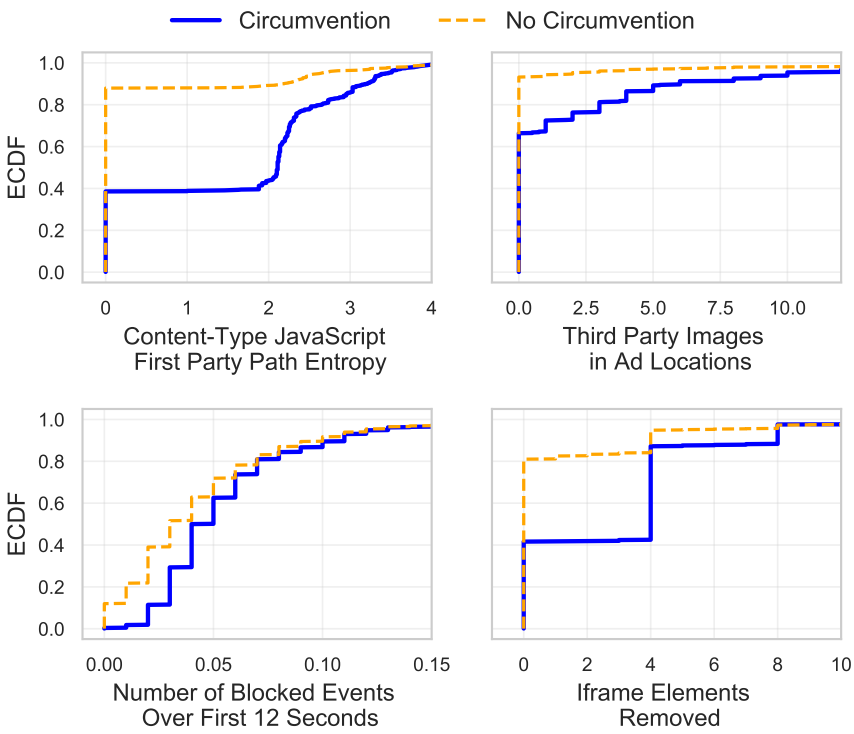


Figure 3.11: **Top-Features ECDF.** We show empirical CDFs of some of the top features for our classifier. JS path entropy is the most discriminatory feature.

Iframe elements removed and third-party images in ad locations are also direct mechanisms of circumvention. The former depicts when sites generally clean up iframes that are being hidden or blocked by the adblocker. The latter details the subsequent actions of re-injecting ad images into previously known ad locations during circumvention. We can infer that if a site completes more ad re-injection actions, then it has a higher chance of circumventing the adblocker. The ECDF of the number of blocked events indicates circumvention, where the adblocker generally blocks more for sites that successfully circumvent the adblocker. This highlights that adblockers do not need to block aggressively for sites where they can easily target the root cause of ads. However, when obfuscation techniques are employed, the adblocker must try harder and has a higher chance of not blocking all ads.

Analysis of Mistakes. Next, we discuss the mistakes made by CV-Inspector and we explain the root causes of false negatives (FN) and false positives (FP).

3.4.5.1 False Negatives (FN)

FN occur when the site circumvented the adblocker but CV-Inspector predicted that it did not. We find that CV-Inspector does not perform well for sites that employ excessive DOM obfuscation. For example, argumentiru.com displays Yandex [145] ads using nested custom HTML tags named <yatag>, while separating the ad image and the ad link in different parts of the ad DOM structure. This makes it hard to identify whether it is an ad or not and to evaluate the ad link for entropy. In addition, strip2.xxx uses MobiAds [96] to display ads with a small square image and the rest is text outside of the image. This differs from regular ads where it is entirely an image with text encapsulated in the image. As a result, CV-Inspector cannot help notify filter list authors when they should update filter rules for these particular cases. However, we argue that CV-Inspector can be extended to cover corner cases to capture CV activity, if the sites are of interest to the adblocker.

Another reason for FN is the logic of triggering circumvented ads for a user. We find that even when a site is capable of circumventing the adblocker, it may choose not to. Though more future work is necessary to infer the business logic of circumvention, we find that for a few cases where the site only triggers circumvention once out of the four times we load the page, CV-Inspector would predict there is no circumvention. However, the classifier confidence is generally higher (~0.40), which is close to a positive label when compared to when a site displays no ads at all within the four page loads.

Lastly, some FNs are due to the limitations of screenshots not conveying whether an ad is first-party or not. Thus, when investigating these sites, we manually go to the sites and found that they were first-party ads and should be labeled as negative. Here, we see that the classifier was able to determine the correct label when it comes to first-party ads.

3.4.5.2 False Positives (FP)

CV-Inspector can mistake sites that heavily rely on affiliation links or third-party links as their own web content. For example, home-made-videos.com comprises completely of links to third-parties with image dimensions that can be considered as ad dimensions. Furthermore, some mistakes by CV-Inspector can be attributed to a site's code mistakes. For instance, when investigating empflix.com, we find that CV-Inspector accurately identifies web requests that correspond to circumvented ad content. However, during re-insertion, the JS errors out because it expects the existence of an element with ID "mewTives" but the container is actually not there. We note that this error does not happen on the site's sub pages where the container does exist, and CV-Inspector correctly predicts that circumvention happens.

We find some false positives were actual true positives but were mislabeled due to the height cap of screenshots. Recall that we limit the height of the screenshots to 3000px to be compatible with sites that would infinitely scroll. We discover that many adult content sites using ExoClick [58] would re-inject ads back near the bottom of the page. We see this as a strength of CV-Inspector that establishes that it can detect circumvention beyond just the top part of the site (*i.e.*, above the fold section [104]).

3.4.6 Feature Robustness

CV-Inspector extracts a diverse range of features that capture different fundamental characteristics of circumvention. We discuss potential approaches that CV providers could utilize to evade each type of feature, along with their effectiveness and trade-offs involved. We argue that it is challenging for CV providers to evade the features used by CV-Inspector, while still achieving their objectives, which are: (1) to evade adblocking rules, to display ads, and to obtain publisher ad revenue; (2) to not degrade the user experience on the publisher's site; and (3) to minimize the cost and overhead incurred by integrating the CV service.

1. Web Request Features. Randomizing URL components, such as subdomains and paths, is a typical obfuscation technique that CV providers use to evade filter rules. However, our entropy features capture not the exact randomized string (which would be easy to evade) but the fact that randomization is used at all (which is robust). An example was shown in Listing 3.2. To bypass these features, a CV provider would have to stop obfuscating URL components altogether, *i.e.*, abandon this circumvention technique.

2. DOM Mutation Features. A CV provider could try to manipulate DOM mutation features. For instance, instead of removing DOM nodes, the provider can hide the nodes. However, circumvention would still be detected by our features relating to "DOM attribute changes," such as display and class. CV providers could also try to add noise by causing dummy DOM mutations. However, unless the provider can affect the "No Adblocker" case as well, it will make circumvention activity even easier to detect via differential analysis. Furthermore, adding too many dummy mutations can make the site slow since the browser must refresh how the page is displayed, which affects the user experience.

3. Temporal Features. The CV provider can try to change the number of blocked elements by making the advertising DOM structure simpler, as shown in Listing 3.3, or more complex by using unnecessary DOM elements. This effectively reduces the number of blocked elements. However, page source features can still detect circumvention by analyzing ad positions rather than the DOM structure. Another exploit is to delay the triggering of circumvention (*e.g.* after the 12-second period) so that CV-Inspector does not detect the number of blocked events. However, this goes against the main objective of ads, which is to quickly display ads to the user before the user leaves the page. This approach would negatively affect the revenue that the publisher wants to recover by employing circumvention in the first place.

4. Page Source Features. To evade the features related to the number of iframes and images in ad positions, a CV provider can change the location of ads when circumvention is em-

ployed. For example, if ads were originally shown on the right sidebar for the "No Adblocker" case, then the ads can be moved to the left sidebar. However, this increases the overhead for the publisher to integrate with CV providers, as the new ad locations must be seamlessly incorporated into individualized templates of different sites. In the above example, the left sidebar must make sense within the publisher's template to be a feasible ad location. Also recall from Fig. 3.1 that the publisher must still fetch for new ad content. Thus, CV-Inspector can still capture this circumvention characteristic through our web request features.

Takeaways. Overall, CV-Inspector raises the bar in the arms race with CV providers, by extracting diverse features that collectively capture the fundamental behavior of CV providers through differential analysis. In order to evade differential analysis, CV providers would have to make the site's behavior "With Adblocker" similar to that of "No Adblocker." However, this either limits ad re-injection to simple static ads (often not profitable for publishers) or requires that CV services are triggered for all users (using adblockers or otherwise) resulting in higher costs for the publisher.

3.4.7 Summary

In this section, we presented the design and implementation of CV-Inspector. Specifically, it collects data from web requests, DOM mutations, temporal information (including blocked events caused by ABP), page source, and screenshots. Then, it employs differential analysis designed uniquely to capture circumvention activity, and we extract intuitive features specifically designed for capturing circumvention. We also provide an iterative methodology for obtaining ground truth, that speeds up the process and discovers more positive labels. We trained and evaluated a Random Forest classifier using this GT dataset, and demonstrated that it achieves an accuracy of 93% in detecting sites that employ CV providers. We further find that web request features relating to path entropy are the most effective features. By capturing the essential characteristics of circumvention, we conclude that it would be difficult

Detection on Tranco-20K Dataset				
Sampling	**Label**	**Predicted**	**Correct**	**Precision**
No	CV	91	79 / 91	87%
Yes	No CV	29,248	345 / 380	91%

Table 3.4: **CV-Inspector on the Tranco-20K.** For "No CV" instances, we sample from that predicted set to have a confidence level of 95% with 5% margin of error.

for CV providers to evade both CV-Inspector and filter rules without incurring costs, *i.e.*, not being able to show profitable ads to the users and overhead of activating circumvention for all users. Next, we apply and evaluate CV-Inspector in real world settings.

3.5 CV-Inspector: In the Wild Deployment

We employ CV-Inspector in two real world scenarios. First, in Sec. 3.5.1, we employ CV-Inspector on the popular Tranco-20K sites to *discover* sites that circumvent adblockers, and are possibly unknown. Second, in Sec. 3.5.2, we use CV-Inspector to *monitor* the effectiveness of ACVL on sites that are well-known to circumvent adblockers, and which are continuously monitored by filter list curators. For the evaluation of monitoring, we use two datasets: our own GTP dataset and Adblock Plus Monitoring dataset provided by ABP. More details are provided in the respective sections and the datasets are detailed in Sec. 3.4.1.4 and Table 3.1.

3.5.1 Discovering Circumvention in the Wild

3.5.1.1 In the Wild Performance

We first conduct a large-scale analysis of deploying CV-Inspector in the wild. Our goal is to facilitate the crowdsourcing effort by the adblocking community to discover sites that successfully circumvent adblockers. To that end, we apply CV-Inspector on the popular Tranco-20K sites, which contains 29.3K pages with sub-pages. Recall that the Tranco top-

2K sites were used as candidates for labeling (CL), which eventually affected the training set (GT) for our CV-Inspector's classifier. Therefore, we exclude it from the in the wild evaluation because we want to keep the Tranco sites used for training (top-2K) and testing (top 2k-20K) disjoint. We follow our earlier data collection approach, described in Fig. 3.6, to crawl these URLs. As shown in Table 3.4, CV-Inspector detects 91 sites as "CV" and the remaining 29,248 sites as "No CV." We validate the 91 "CV" sites and a random sample (380) of "No CV" sites. CV-Inspector achieves 87% precision when identifying sites with successful CV and 91% for the opposite case. Our evaluation in Table 3.4 shows that CV-Inspector generalizes well in the wild, with similar precision compared to Table 3.3.

The Random Forests classifier picks the likeliest class, which in binary classification is by default the class with a probability above 0.5. This is the case in the results presented in Tables 3.4, 3.6, and 3.7 in this section. CV-Inspector can be applied to different use cases (*e.g.* discovery or monitoring of sites employing circumvention) that value different metrics (*e.g.* recall vs. precision, respectively). Since there is no universally applicable operating point, instead of tuning parameters to overfit a particular use case and dataset, we discuss the trade-offs involved and leave it up to the users of CV-Inspector to decide upon the operating point that matches their goals.

Trade-offs. Fig. 3.12 reports how CV-Inspector navigates the trade-off between discovering more sites that successfully circumvent adblockers and precision, when applied to the Tranco-20K dataset. For instance, a confidence level threshold of 0.6 achieves a precision of 98% with only one FP. This would be an attractive option to minimize human supervision for monitoring sites of interest. However, if discovering sites that use circumvention is more important, then lowering the threshold below 0.5 would find more sites at the expense of increasing human efforts to deal with FPs. The operating point can be tweaked to optimize various objectives of interest.

As a concrete example, Fig. 3.13(a) depicts how CV-Inspector can navigate the trade-

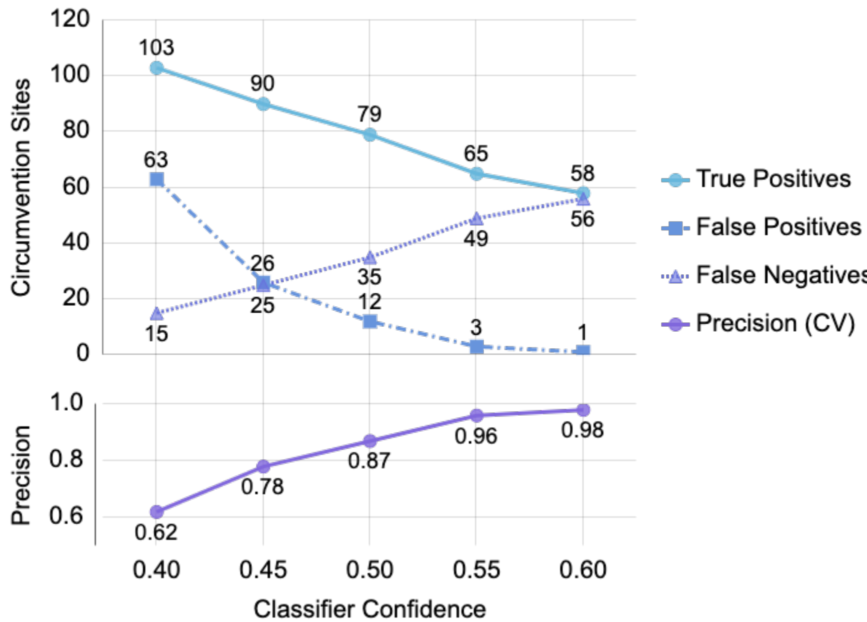
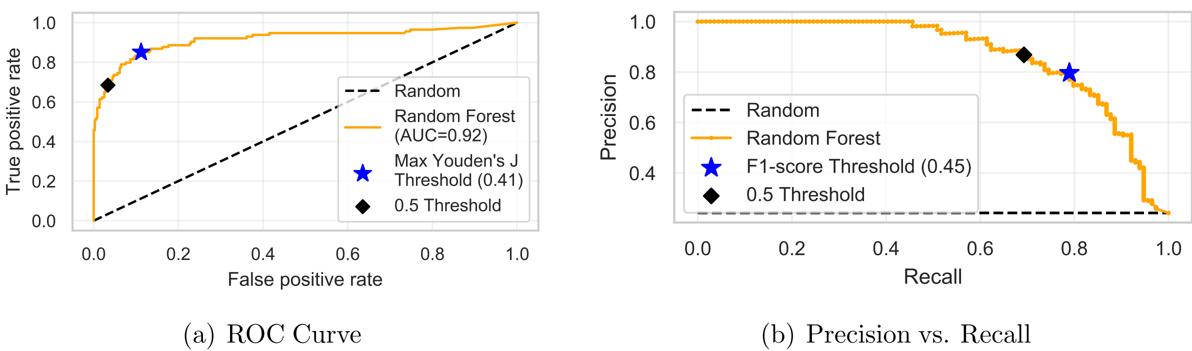


Figure 3.12: **Discovery vs. Precision.** The trade-off between discovering more circumvention sites (positive instances) within our Tranco-20K (in the wild) dataset *vs.* being correct in the prediction.



(a) ROC Curve

(b) Precision vs. Recall

Figure 3.13: **Trade-offs on the Tranco-20K Dataset. (a)** Within our ROC curve, we can maximize Youden's J index if we value a high true positive rate (TPR) with a low false positive rate (FPR) for the purpose of discovering sites with successful circumvention. The threshold following these criteria is 0.41, which corresponds to a TPR of 0.85 and FPR of 0.11. **(b)** Within our precision-recall curve, we can find the threshold that corresponds to the optimal F1-score for positive labels. The threshold following these criteria is 0.45, which achieves an F1-score of 0.79.

off between true positive rate (TPR) and false positive rate (FPR). The suitable classifier threshold depends on the use case. For example, if one wants to optimize for TPR (*i.e.*, recall) while keeping the FPR low, one metric to maximize is Youden's J index, which leads

to a threshold of 0.41 with a corresponding TPR of 0.85 and an FPR of 0.11. This is the right objective when we are interested in discovering more sites that employ circumvention at the risk of some additional false positives.

Another even more relevant trade-off in our case is precision vs. recall, depicted in Fig. 3.13(b). We find that a threshold of 0.45 maximizes the F1-score for positive labels, achieving an F1-score of 0.79. It is not surprising that this value is close but below 0.5, because our Tranco-20K dataset is imbalanced as shown in Table 3.4. As discussed in Sec. 3.4.4, positive labels are rare when compared to negative labels, which makes the classifier less sensitive to the minority class. To compensate for this, one would decrease the threshold to improve recall for positive labels at the expense of precision.

3.5.1.2 Circumvention Providers

We now analyze the breakdown of different circumvention providers. Note that CV-Inspector is not designed to distinguish between different circumvention providers. Therefore, we rely on other heuristics to detect specific CV providers.

Unique Keywords for CV Providers. We curate keywords that are indicative of specific CV providers through a careful manual inspection of known circumvention sites in our GT dataset. Intuitively, to discover keywords, we first search using the name of the providers within our collected data of web requests (*e.g.*, URLs and HTTP request/response headers) and page source files (*e.g.*, HTML files consisting of HTML, inline CSS, and inline JS). Notably, we discover that some CV providers, like ExoClick, AdDefend, and Adthrive, do not attempt to hide their presence, as the name of the provider was sufficient to be used as keywords. For example, we found that ExoClick can be detected by the keywords “exoclick” and “exoloader” in the page source. For AdDefend, AdThrive and MobiAds, we look at keywords “addefend,” “adthrive,” and “mobiads” in the page source, respectively. Similarly, Publica can be detected by looking for the key ”publica_user_id” in the HTTP response

CV Providers	Count	WR	DOM	CV-Inspector precision
AdThrive [18]	154	●	◐	98%
Publica [112]	77	●	◐	95%
ExoClick [58]	76	◐	◐	100%
Yandex [145]	434	◐	●	100%
AdDefend [11]	38	◐	●	N/A
MobiAds [96]	17	◐	●	N/A

Table 3.5: **Circumvention Providers and Approaches**. We show the presence of circumvention providers within the Tranco-20K. We use ● to mean full obfuscation, which means randomized URL components (WR) or deeply nested nonstandard DOM structures for ad (DOM). ◐ denotes partial obfuscation, which means ad resources may be hidden with first-party domain (WR) and ad reinsertion uses simpler DOM structures (DOM). WR = Webrequests, DOM = DOM changes.

Set-Cookie header. When the name of the provider was not enough, we inspected the DOM structure of the ad using the page source to see if there was any unique identifier that we could use. In this case, we find that Yandex can be detected by looking for the custom DOM tag "<yatag>." In total, we utilized seven keywords to identify the presence of six different CV providers listed in Table 3.5.

We note that the presence of these keywords does not necessarily always indicate that circumvention was successful. It could also mean that the circumvention attempt failed or that circumvention was not even attempted (e.g., dormant code). Therefore, we cannot simply use these heuristics in place of CV-Inspector to detect sites that circumvent adblockers. Table 3.5 summarizes the application of the aforementioned heuristics on Tranco-20K sites. We identify many instances of different CV providers, including ad networks such as Yandex and dedicated CV providers such as AdThrive and AdDefend.

Taxonomy of Circumvention Approaches. Next, we characterize the obfuscation approaches used by different CV providers by defining whether the obfuscation is full (●) or partial (◐). For web request obfuscation, full obfuscation refers to the use of randomized URLs including subdomains and paths as shown in Listing 3.2 while partial obfuscation

refers to the use of first-party subdomains. For DOM obfuscation, full obfuscation refers to the use of non-standard DOM structures such as deeply nested elements or randomized tag attributes while partial obfuscation refers to only randomized tag attributes with simple DOM structures, such as Listing 3.3.

Using this taxonomy, we compare the full *vs.* partial obfuscation techniques of different CV providers. First, ExoClick and AdDefend simply leverage inlined JS, which is difficult to block without hurting other page functionality [42], to implement their circumvention logic. AdThrive redirects through several domains (*e.g.*, cloudfront.net → edvfwlacluo.com → lmyiwaakn.com) before fetching the JS that implements their circumvention logic. Second, ExoClick and AdDefend do not obfuscate URLs but rather serve their ad resources under first-party domains that are difficult to distinguish from legitimate content. AdThrive fetches ads in iframes using rotating third-party domains, subdomains, and randomized IDs. Third, ExoClick and AdDefend differ in their DOM obfuscation techniques. ExoClick uses a simpler ad structure (a hyperlink with two div children) while obfuscating the ad image by serving it with CSS background-image instead of a regular image tag. On the other hand, AdDefend employs a nested DOM structure with obfuscated IDs, while Yandex uses nested non-standard tags with obfuscated class names.

Finally, we analyze CV-Inspector's performance in detecting different CV providers. We match each detected CV provider instance in Table 3.5 to our CV-Inspector deployment results on Tranco-20K sites from Table 3.4. We see that CV-Inspector achieves good precision in detecting different popular CV providers. For AdDefend and MobiAds, we use "N/A" to denote that we lack sufficient data.

Ground Truth Positives (GTP) Dataset				
Sampling	**Label**	**Predicted**	**Correct**	**Precision**
No	CV	244	223 / 244	91%
Yes	No CV	465	187 / 211	89%

Table 3.6: **CV-Inspector on the GTP Dataset.** We show the results of applying our classifier on the ~700 sites from our ground truth that also originated from ACVL (Table 3.1). However, this time we collect the data by turning on ACVL as well within our custom ABP extension. For "No CV" instances, we sample from that predicted set to have a confidence level of 95% with a 5% margin of error.

Adblock Plus Monitoring Dataset				
Sampling	**Label**	**Predicted**	**Correct**	**Precision**
No	CV	5	4 / 5	80%
Yes	No CV	355	184 / 185	99%

Table 3.7: **CV-Inspector on the Adblock Plus Monitoring Dataset.** From a real world dataset used by ABP to monitor circumvention, we apply our classifier and show the results. For "No CV" instances, we sample from that predicted set to have a confidence level of 95% with a 5% margin of error.

3.5.2 Monitoring Circumvention for Sites of Interest

As discussed in Sec. 3.3.2, ACVL is updated very frequently to combat the back and forth between adblockers and circumvention providers. In addition, filter list authors generally focus their attention on "sites of interest," as discussed in Sec. 3.4.1.4. Curators must continuously monitor them to see if the filter list (ACVL) continues to be effective, or if circumvention has evolved, and the filter rules need updating. Consequently, much human labor goes to this continuous monitoring of sites in the ACVL. To that end, we show how CV-Inspector can automatically monitor whether ACVL is effective in countering circumvention on a site. We use the same approach as laid out in Fig. 3.6 but with one change. We use the ACVL, in addition to EL, when crawling a site with an adblocker. We use two datasets from Table 3.1 for evaluation: (1) the GTP dataset, which contains all sites that circumvent the adblocker in our GT; and (2) Adblock Plus Monitoring dataset, which contains 360 sites that ABP continuously monitor for circumvention activity to update filter rules.

3.5.2.1 Monitoring sites in GTP

We use CV-Inspector to classify sites within our GTP dataset, which comprises of 700 sites from the GT dataset that originated from ACVL and were successful at circumventing the adblocker. If CV-Inspector again detects a site as "CV," it shows that the site is able to successfully circumvent even the ACVL. We manually validate CV-Inspector's classifications. Table 3.6 summarizes the results. We note that CV-Inspector again detects 244 sites as "CV" with 91% precision and 465 sites as "No CV" with 89% precision. The results show that more than one-third of sites with relevant filter rules in the ACVL are still able to successfully circumvent adblockers. This demonstrates that the sites addressed by the ACVL need to be continuously monitored. We suggest that CV-Inspector should be periodically used (e.g., every hour) to monitor the sites on the ACVL. The sites that are detected by CV-Inspector would need to be reviewed by ACVL curators to update the filter rules and the rest can be safely ignored.

3.5.2.2 Monitoring sites from ABP

To further demonstrate CV-Inspector's usefulness, we obtain a list of 360 sites from ABP that are manually monitored by the ABP team due to the sites' fast-paced adaptation to changes in the ACVL. Table 3.7 summarizes the results of applying CV-Inspector (with ACVL) on these sites. Out of these sites, we note that 5 sites are detected as "CV" and the remaining 355 as "No CV," again with high precision. This finding shows that even the sites that are closely monitored to be addressed by the ACVL team can successfully circumvent the adblocker. Notably, if we consider only the 190 sites that we labeled as human labeling effort, then CV-Inspector was able to save up to 98% of the work for ACVL curators by predicting 188 sites correctly. Thus, CV-Inspector can help with continuously monitoring these sites at a high frequency.

3.6 Discussion and Future Directions

Summary. In this paper, we studied an emerging threat in the advertising ecosystem: circumvention (CV) services that help publishers bypass adblockers and re-injects ads. CV services are sophisticated, opaque for the user, and exploit fundamental weaknesses of adblockers' design and the open-source nature of anti-CV community efforts (exemplified by the anti-CV list). Although there has been increasing anecdotal evidence about the adoption of circumvention in the wild, to the best of our knowledge, ours is the first large-scale study of the state of the circumvention arms race. We develop CV-Inspector: a methodology for automatically crawling sites of interest and a classifier that can accurately detect whether a site successfully circumvents the adblocker or not. We envision that CV-Inspector will serve as an automation tool for filter list curators to help them focus their inspection efforts on discovering new sites that employ circumvention in the wild and in monitoring sites of interest continuously in the arms race between circumvention and anti-CV filter rules.

Open Source Tools. We plan to make CV-Inspector available to the community at [82]. This will include the datasets (including our labeled dataset of top-20K crawled sites), crawling instrumentation (shareable as Amazon Machine Images [21]), and the trained classifier.

Limitations. There are limitations in our design and implementation. First, CV-Inspector uses differential analysis that relies on differences between the "No Adblocker" and "With Adblocker" cases. If sites exhibit no actual differences in the two cases, then CV-Inspector will not be able to detect circumvention. For example, searchenginereports.net already includes circumvented ads in the DOM structure of the "No Adblocker" case but only hidden. When it detects an adblocker affecting its ads, it will simply show the backup ads that were already there. Second, CV-Inspector only considers circumvention that appears without user interaction. For instance, shahid4u.cam displays no visual ads to the user, but when the user clicks on a link, it will redirect the user to an ad before showing the real content.

More details on implementation choices and limitations are provided in Sec. 3.4.1.3.

Future Directions. We plan to further *automate filter rule generation* and help anti-CV authors, by building on two opportunities already provided by CV-Inspector. First, our differential analysis already uncovers web requests that are related to circumvention. Consider the spring.org.uk example: CV-Inspector already pinpoints all randomized paths and subdomains of podfdch.com. Using that information, a filter list author can simply create a filter rule such as "*.podfdch.com^" or any variations of its subdomains and paths if there are common prefixes and suffixes like "||podfdch.com/erej*". Second, our feature extraction already dynamically generates CSS selectors of ad locations where re-injection can happen. Filter list authors can translate them into DOM element hiding rules, as described in Table 2.2. They can infer the effectiveness of the selectors — the more elements that match, the more ads the selectors will affect.

It also remains to be seen how robust CV-Inspector is in the presence of ever-changing circumvention obfuscation techniques. Our intuition is that the features used by CV-Inspector (*e.g.*, randomness in an obfuscated path) are inherently more long-lived and harder to evade than the exact rules used by filter lists (*e.g.*, the actual randomized string in the path). It would be interesting to characterize the *time scales* of this arms race.

Feature engineering can also be improved. We can consider new features (*e.g.* extracted from JS) and improve existing features (*e.g.* the way we capture DOM mutation, by taking into account the DOM graph structure in the differential analysis). With respect to JS in particular, the current version of CV-Inspector does not take into account JS features on purpose, because CV providers heavily obfuscate JS, which makes differential analysis challenging. As shown in Fig. 3.1, this involves retrieving new ad content (web requests) and displaying the ad to the user at the end (DOM structure). The technique that JS utilizes to re-inject ads back upon the page does not matter: as long as CV-Inspector can recognize the final DOM structure, it can still detect circumvention.

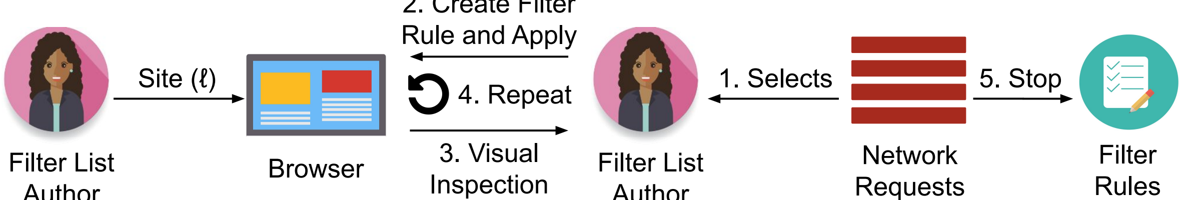
Overall, we consider CV-Inspector to be the first significant step towards automating aspects of the defense (ad blockers, filter list authors' effort) against circumvention by showing that it can reduce human labeling efforts by 98%. The longer term goal is to fully automate the defense against circumvention through detection and filter rule generation. In Chapter 4, we further address filter rule generation by developing a framework to generate URL-based filter rules using reinforcement learning.

Chapter 4

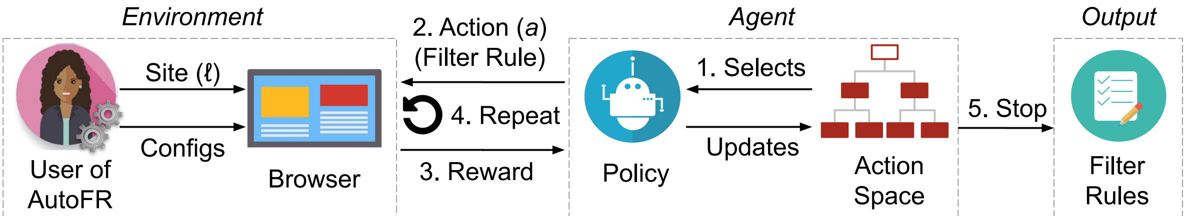
AutoFR: Automated Filter Rule Generation for Adblocking

4.1 Introduction

As discussed in Sec. 2.1.2, adblockers are prevalent, especially on the web, and depend on filter rules for their adblocking functionality. It is well-known that the filter rule curation process is slow and error-prone [19], and requires significant continuous effort by the filter list community to keep them up-to-date [85]. As a result, as described in Sec. 2.1.3, the research community is actively working on machine learning (ML) approaches to assist with filter rule generation [30, 66, 124] or to build models to replace filter lists altogether [73, 122, 146, 2]. There are two key limitations of prior ML-based approaches. First, existing ML approaches are supervised as they rely on human feedback and/or existing filter lists (which are also manually curated) for training. This introduces a circular dependency between these supervised ML models and filter lists — the training of models relies on the very filter lists (and humans) that they aim to augment or replace. Second, existing ML approaches do not explicitly consider the trade-off between blocking ads and avoiding break-



(a) **Filter List Authors' (Human) Workflow.** How filter list authors create filter rules for a site ℓ: (1) they select a network request caused by the site; (2) they create a rule and apply it on the site; (3) they visually inspect whether it blocked ads without breakage; (4) they repeat the process if necessary for other network requests; and (5) they stop when they have crafted rules that can block all/most ads for the site without causing significant breakage.



(b) **AutoFR (Automated) Workflow.** AutoFR automates these steps as follows: (1) the agent selects an action (*i.e.,* filter rule) following a policy; (2) it applies the action on the environment; (3) the environment returns a reward, used to update the action space; (4) the agent repeats the process if necessary; and (5) the agent stops when a time limit is reached, or no actions are available to be explored. The human filter list author only provides a site ℓ and configurations (*e.g.,* threshold w and hyper-parameters).

Figure 4.1: AutoFR automates the steps taken by FL authors to generate filter rules for a particular site. FL authors can configure the AutoFR parameters but no longer perform the manual work. Once rules are generated by AutoFR, it is up to the FL authors to decide when and how to deploy the rules to end-users.

age. An over-aggressive adblocking approach might block all ads on a site but may block legitimate content at the same time. Thus, despite recent advances in ML-based adblocking, filter lists remain defacto in adblocking.

Fig. 4.1(a) illustrates the workflow of a FL author for creating rules for a particular site: (1) select a network request to block; (2) design a filter rule that corresponds to this request and apply it on the site; (3) visually inspect the page to evaluate if the filter rule blocks ads and/or causes breakage and; (4) repeat for other network requests and rules; since modern sites are highly dynamic, and often more so in response to adblocking [85, 19, 149, 43], the FL author usually revisits the site multiple times to ensure the rule remains effective; and (5) stop when a set of filter rules can adequately block ads without causing breakage.

We ask the question: *how can we minimize the manual effort of FL authors by automating the process of generating and evaluating adblocking filter rules?* We propose AutoFR to automate each of the aforementioned steps, as illustrated in Fig. 4.1(b), and we make the following contributions.

First, we formulate the filter rule generation problem within a reinforcement learning (RL) framework, which enables us to efficiently create and evaluate good candidate rules, as opposed to brute force or random selection. We focus on URL-based filter rules that block ads, a popular and representative type of rules that can be visually audited. An important component, which replaces the visual inspection, is the detection of ads (through a perceptual classifier, Ad Highlighter [127]) and of visual breakage (through JavaScript [JS] for images and text) on a page. We design a reward function that combines these metrics to enable explicit control over the trade-off between blocking ads and avoiding breakage.

Second, we design and implement AutoFR to train the RL agent by accessing sites in a controlled realistic environment. It creates rules for a site in under two minutes, which is crucial for scalability. We provide in-depth details of our implementation, hyper-parameter tuning, and experimental setup for future researchers to reproduce and extend AutoFR. We deploy and evaluate AutoFR's efficient implementation on Top–5K websites, and we find that the filter rules generated by AutoFR block 86% of the ads. The effectiveness of the AutoFR rules is overall comparable to EasyList in terms of blocking ads and visual breakage.

Third, we address how to generate filter rules that are applicable across multiple sites. We leverage AutoFR as a building block to design three new approaches. The first modifies our AutoFR algorithm to accept a set of sites and optimize rules for them. The second utilizes the notion of rule popularity, *i.e.,* if the same rules are created individually for several sites, then intuitively, they will also be good for other sites. The third relies on collaborative filtering, *i.e.,* use existing rules from the known sites and apply them to other sites based on site similarity. We compare these approaches against AutoFR as our baseline using

both controlled and in the wild experiments and evaluate them for effectiveness, efficiency, and maintainability. We envision that the adblocking community will use AutoFR and its variations to automatically generate and update filter rules at scale.

The rest of our paper is organized as follows. Sec. 4.2 provides background and related work. Sec. 4.3 formalizes the problem of filter rule generation, including the human process, the formulation as an RL problem, and our particular multi-arm bandit algorithm for solving it. Sec. 4.4 presents our implementation of the AutoFR framework. Sec. 4.5 provides its evaluation on the Top–5K sites. Sec. 4.6 describes and compares new approaches to generating filter rules across multiple sites. Sec. 4.7 provides our implementation of AutoFR in a live setting. Sec. 4.8 concludes the paper.

4.2 Most Closely Related Work

Reinforcement Learning. We formulate the problem of filter rule curation *from scratch* (*i.e.,* without any ground truth or existing list) as a reinforcement learning (RL) problem; see Sec. 4.3. Within the vast literature in RL [128], we choose the Multi-Arm Bandits (MAB) framework [22], for reasons explained in Sec. 4.3.2. Identifying the top–k arms [38, 89] rather than searching for the one best arm [60] has been used in the problems of coarse ranking [76] and crowd-sourcing [40, 68]. Contextual MAB has been used to create user profiles to personalize ads and news [87]. Bandits where arms have similar expected rewards, commonly called Lipschitz bandits [77], have also been utilized in ad auctions and dynamic pricing problems [78]. In our context of filter rule generation, we leverage the theoretical guarantees established for MAB to search for "good" filter rules and identify the "bad" filter rules, while searching for opportunities of "potentially good" filter rules (hierarchical problem space [141]), as discussed in Sec. 4.3.3. While RL algorithms, in general, have been applied to several application domains [56, 147, 33, 57], RL often faces challenges in the real-world [53] including convergence and adversarial settings [62, 144, 114, 71, 23].

Our Work in Perspective. The design of the framework is described in Sec. 4.3 and illustrated in Fig. 4.1(b). AutoFR is the first to fully automate the process of filter rule generation and create URL-based, per-site rules that block ads from scratch, using reinforcement learning. The majority of prior ML-based techniques relied on existing filter lists at some point in their pipeline, thus creating a circular dependency. Furthermore, AutoFR is the first to choose the granularity of the URL-based rule to explicitly optimize the trade-off between blocking ads and avoiding visual breakage.

The implementation is described in Sec. 4.4 and illustrated in Fig. 4.3. Within the RL framework, AutoFR's key design contributions include the action space, the RL components (*e.g.,* agent, environment, reward, policy), the annotation of raw AdGraphs into site snapshots, and the logic and implementation of utilizing site snapshots to emulate site visits. The latter was instrumental in scaling the approach (it reduced the time for generating rules for a single site from approximately 13 hours to 1.6 minutes) and making our results reproducible. For some individual RL components, we leverage state-of-the-art tools: (1) we utilize one part of AdGraph that creates a graph representing the site (we do *not* use the trained ML model of AdGraph); and (2) we use Ad Highlighter to automatically detect ads, which is used to compute our reward function. As these individual components improve over time, the AutoFR framework can benefit from new and improved versions or even incorporate newly available tools in the future.

4.3 The AutoFR Framework

In this section, we formalize the problem of filter rule generation, including the process followed by human FL authors (Sec. 4.3.1 and Fig. 4.1(a)), our formulation as a reinforcement learning problem (Sec. 4.3.2 and Fig. 4.1(b)), and our multi-arm bandit algorithm for solving it (Sec. 4.3.3 and Alg. 1).

	Description	Filter Rule
1	eSLD	$\|\|ad.com\hat{}$
2	FQDN	$\|\|img.ad.com\hat{}$
3	With Path	$\|\|ad.com/banners/$ or $\|\|img.ad.com/banners/$

Table 4.1: **URL-based Filter Rules.** They block requests, listed from coarser to finer-grain: eSLD (effective second-level domain), FQDN (fully qualified domain), With Path (domain and path). Other types of filter rules are provided in Table 2.2.

4.3.1 Filter List Authors' Workflow

Scope. Among all possible filter rules, we focus on the important case of *URL-based rules for blocking ads* to demonstrate our approach. Table 4.1 shows examples of URL-based rules at different granularities: blocking by the effective second-level domain (eSLD), fully qualified domain (FQDN), and including the path.

Filter List Authors' Workflow for Creating Filter Rules. Our design of AutoFR is motivated by the bottlenecks of filter rule generation, revealed by prior work [85, 19], our discussions with FL authors, and our own experience in curating filter rules. Next, we break down the process that FL authors employ into a sequence of tasks, also illustrated in Fig. 4.1(a). When FL authors create filter rules for a specific site, they start by visiting the site of interest using the browser's developer tools. They observe the outgoing network requests and create, try, and select rules through the following workflow.

Task 1: *Select a Network Request*. FL authors consider the set of outgoing network requests and treat them as candidates to produce a filter rule. The intuition is that blocking an ad request will prevent the ad from being served. For sites that initiate many outgoing network requests, it may be time-consuming to go through the entire list. When faced with this task, FL authors depend on sharing knowledge of ad server domains with each other or heuristics based on keywords like "ads" and "bid" in the URL. FL authors may also randomly select network requests to test.

Task 2: *Create a Filter Rule and Apply***.** FL authors must create a filter rule that blocks the selected network request. However, there are many options to consider since rules can be the entire or part of the URL, as shown in Table 4.1. FL authors intuitively handle this problem by trying first an eSLD filter rule because the requests can belong to an ad server (*i.e.,* all resources served from the eSLD relate to ads). However, the more specific the filter rule is (*e.g.,* eSLD $\rightarrow$ FQDN), the less likely it would lead to breakage. Then, the FL authors apply the filter rule of choice onto the site.

Task 3: *Visual Inspection***.** Once the rule is applied on the site, FL authors inspect its effect, *i.e.,* whether it indeed blocks ads and/or causes breakage (*i.e.,* legitimate content goes missing or the page displays improperly). FL authors use differential analysis. They visit a site with and without the rule applied, and they visually inspect the page and observe whether ads and non-ads (*e.g.,* images and text) are present/missing before/after applying the rule. In assessing the effectiveness of a rule, it is essential to ensure that it blocks at least one request, *i.e., a hit.* Filter rules are considered "good" if they block ads without breakage and "bad" otherwise. Avoiding breakage is critical for FL authors because rules can impact millions of users. If a rule blocks ads but causes breakage, it is considered a "potentially good" rule.

Task 4: *Repeat***.** FL authors repeat the process of Tasks 1, 2, 3, multiple times to make sure that the filter rule is effective. Repetition is necessary because modern sites typically are dynamic. Different visits to the same site may trigger different page content being displayed and different ads being served. If a rule from Task 2 blocks ads but causes breakage, the author may try a more granular filter rule (*e.g.,* eSLD $\rightarrow$ FQDN from Table 4.1). If the rule does not block ads, go back to Task 1.

Task 5: *Stop and Store Good Filter Rules***.** FL authors stop this iterative process when they have identified a set of filter rules that block most ads without breakage (*i.e.,* a best-effort approach). None of the considered rules may satisfy these (somewhat subjective) conditions, in which case no filter rules are produced.

Bottlenecks: Scale and Human-in-the-Loop. The workflow above is labor-intensive and does not scale well. There is a large number of candidate rules to consider for sites with a large number of network requests (Task 1) and long and often obfuscated URLs (Task 2). The scale of the problem is amplified by site dynamics, which requires repeatedly visiting a site (Task 4). The effect of applying each single rule must then be evaluated by the human FL author through visual inspection (Task 3), which is time-consuming on its own.

Motivated by these observations, we aim to automate the process of filter rule generation per-site. We reduce the number of iterations needed (by intelligently navigating the search space for good filter rules via reinforcement learning), and we minimize the work required by the human FL author in each step (by automating the visual inspection and assessment of a rule as "good" or "bad"). Our proposed methodology is illustrated in Fig. 4.1(b) and formalized in the next section.

4.3.2 Reinforcement Learning Formulation

As illustrated in Fig. 4.1(a), FL authors repeatedly apply different rules and evaluate their effects until they build confidence on which rules are generally "good" for a particular site. This repetitive action-response cycle lends itself naturally to the *reinforcement learning (RL)* paradigm, as depicted in Fig. 4.1(b), where actions are the applied filter rules and rewards (response) must capture the effectiveness of the rules upon applying them to the site (environment). Testing all possible filter rules by brute force is infeasible in practice due to time and power resources. However, RL can enable efficient navigation of the action space.

Specifically, we choose the *multi-arm bandit (MAB)* RL formulation. The actions in MAB are independent *k-bandit arms* and the selection of one arm returns a numerical reward sampled from a stationary probability distribution that depends on this action. The reward determines if the selected arm is a "good" or a "bad" arm. Through repeated action selection, the

objective of the MAB agent is to maximize the expected total reward over a time period [22].

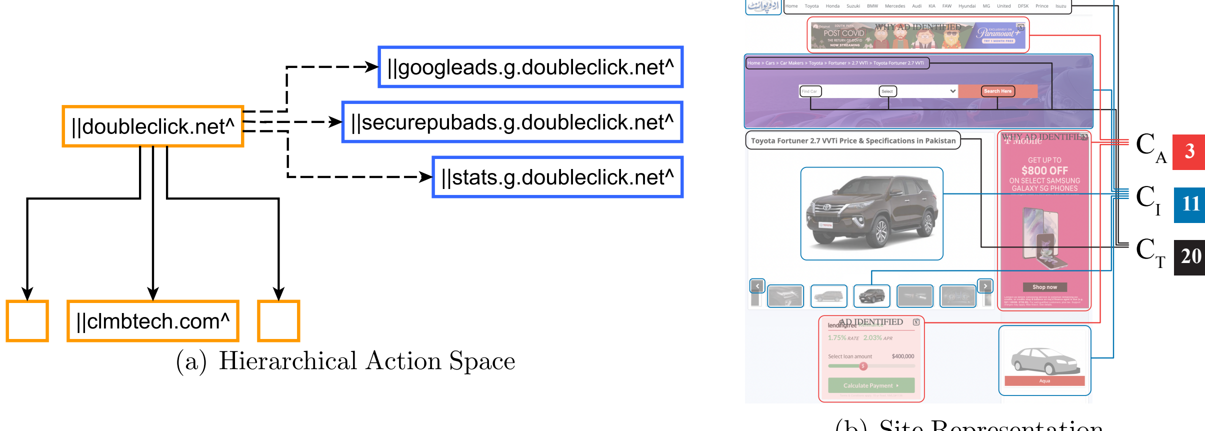
The MAB framework fits well with our problem. The *MAB agent* replaces the human (FL author) in Fig. 4.1(a). The agent knows all available "arms" (possible filter rules), *i.e.,* the action space; see Sec. 4.3.2.1. The agent picks a filter rule (arm) and applies it to the *MAB environment*, which, in our case, consists of the site ℓ (with its unknown dynamics as per Task 4), the browser, and a selected configuration (how we value blocking ads vs. avoiding breakage, explained in Sec. 4.3.3). The latter affects the reward of an action (rule) the agent selects. Filter rules are independent of each other. Furthermore, the order of applying different filter rules does not affect the result. For adblockers, like Adblock Plus, blocking rules do not have precedence. Through exploring available arms, the agent efficiently learns which filter rules are best at blocking ads while minimizing breakage; see Sec. 4.3.2.2. Next, we define the key components of the proposed AutoFR framework, depicted in Fig. 4.1(b). It replaces the human-in-the-loop in two ways: (1) the FL author is replaced by the MAB policy that avoids brute force and efficiently navigates the action space; and (2) the reward function is automatically computed, as explained in Sec. 4.3.2.2, without requiring a human's visual inspection.

4.3.2.1 Actions

An *action* is a URL blocking filter rule that can have different granular levels, shown in Table 4.1, and is applied by the agent onto the environment. We use the terms action, arm, and filter rule, interchangeably.

Hierarchical Action Space $\mathcal{A}_{\mathbf{H}}$. Based on the outgoing network requests of a site ℓ (Task 1), there are many possible rules that can be created (Task 2) to block that request. Fig. 4.2(a) shows an example of dependencies among candidate rules:

1. We should try rules that are coarser grain first before trying more finer-grain rules (the horizontal dotted lines). In other words, try *doubleclick.net*, then *stats.g.doubleclick.net*.



(a) Hierarchical Action Space

(b) Site Representation

Figure 4.2: **(a) Hierarchical Action Space.** A node (filter rule) within the action space has two different edges (*i.e.*, dependencies to other rules): (1) the initiator edge, →, denotes that the source node initiated requests to the target node; and (2) the finer-grain edge, - - →, targets a request more specifically, as discussed in Task 4 and Table 4.1. **(b) Site Representation.** We represent a site as counts of visible ads (C_A), images (C_I), and text (C_T), as explained in Sec. 4.3.2.2. Applying a filter rule changes them, by blocking ads (reducing C_A) and/or hiding legitimate content (changing C_I and C_T, thus breakage $\mathcal{B}$).

This intuition was discussed in Task 4.

2. If *doubleclick.net* initiates requests to *clmbtech.com*, we should explore it first, before trying *clmbtech.com* (the vertical solid lines). Sec. 4.4.2 describes how we retrieve the initiator information.

The dependencies among rules introduce a hierarchy in the *action space* $\mathcal{A}_H$, which can be leveraged to expedite the exploration and discovery of good rules via pruning. If an action (filter rule) is good (it brings a high reward, as defined in Sec. 4.3.2.2), the agent no longer needs to explore its children. We further discuss the size of action spaces in Fig. 4.6; we show that they can be large. The creation of $\mathcal{A}_H$ automates Task 2.

4.3.2.2 Rewards

Once a rule is created, it is applied on the site (Task 2). The human FL author visually inspects the site, before and after the application of the rule, and assesses whether ads have

been blocked without breaking the page (Task 3). To automate this task, we need to define a reward function for the rule that mimics the human FL author's assessment of whether a rule blocks ads and the breakage that could occur.

Site Representation. We abstract the representation of a site ℓ by counting three types of content visible to the user: we count the ads (C_A), images (C_I), and text (C_T) displayed. An example is shown in Fig. 4.2(b). The *baseline representation* refers to the site before applying the rule. Since a site ℓ has unknown dynamics (Task 4), we need to visit it multiple times and average these counters: $\overline{C}_A$, $\overline{C}_I$, and $\overline{C}_T$.

We envision that obtaining these counters from a site can be done not only by a human (as it is the case today in Task 3) but also automatically using image recognition (*e.g.,* Ad Highlighter [127]) or better tools as they become available. This is an opportunity to remove the human-in-the-loop and further automate the process. We further detail this in Sec. 4.4.3.

Site Feedback after Applying a Rule. When the agent applies an action a (rule), the site representation will change from ($\overline{C}_A$,$\overline{C}_I$,$\overline{C}_T$) to (C_A, C_I, C_T). The intuition is that after applying a filter rule, it is desirable to see the number of ads decrease as much as possible (ideally $C_A = 0$) and continue to see the legitimate content (*i.e.,* no change in C_I, C_T compared to the baseline). To measure the difference before and after applying the rule, we define the following:

$$\widehat{C}_A = \frac{\overline{C}_A - C_A}{\overline{C}_A}, \quad \widehat{C}_I = \frac{|\overline{C}_I - C_I|}{\overline{C}_I}, \quad \widehat{C}_T = \frac{|\overline{C}_T - C_T|}{\overline{C}_T} \tag{4.1}$$

$\widehat{C}_A$ measures the fraction of ads blocked; the higher, the better the rule is at *blocking ads*. Ideally all ads are blocked, *i.e.,* $\widehat{C}_A$ is 1. In contrast, $\widehat{C}_I$ and $\widehat{C}_T$ measure the fraction of page broken. Higher values incur more breakage. $\widehat{C}_A$, $\widehat{C}_I$ and $\widehat{C}_T$ are bounded between [0,1]. Next, we define *page breakage* ($\mathcal{B}$) as the visible images ($\widehat{C}_I$) and text ($\widehat{C}_T$), which are *not*

related to ads but are missing after a rule is applied:

$$\mathcal{B}=\frac{\widehat{C}_I+\widehat{C}_T}{2} \tag{4.2}$$

We take a neutral approach and treat both visual components equally and average $\widehat{C}_I$, $\widehat{C}_T$. This can be configured to express different preferences by the user, *e.g.,* treat content above-the-fold as more important. Lastly, *avoiding breakage* is measured by $1-\mathcal{B}$. It is desirable that $1-\mathcal{B}$ is 1, and the site has no visual breakage.

Trade-off: Blocking Ads ($\widehat{C}_A$) vs. Avoiding Breakage ($1-\mathcal{B}$). The goal of a human FL author is to choose filter rules that block as many ads as possible (high $\widehat{C}_A$) without breaking the page (high $1-\mathcal{B}$). There are different ways to capture this trade-off. We could have taken a weighted average of $\widehat{C}_A$ and $\mathcal{B}$. However, to better mimic the practices of today's FL authors, we use a *threshold* $w\in[0,1]$ as a design parameter to control how much breakage a FL author tolerates: $1-\mathcal{B}\geq w$. Blocking ads is easy when there is no constraint on breakage — one can choose rules that break the whole page. FL authors control this either by using more specific rules (*e.g.,* eSLD → FQDN) to avoid breakage or avoid blocking at all. We rely on this trade-off as the basis of our evaluation in Sec. 4.5. It is desirable to operate where $\widehat{C}_A=1$ and $1-\mathcal{B}=1$. In practice, FL authors tolerate little to no breakage, *e.g.,* $w\geq 0.9$. However, w is a configurable parameter in our framework.

Reward Function $\mathcal{R}_\mathbf{F}$. When the MAB agent applies a filter rule F (action a) at time t on the site ℓ (environment), this will lead to ads being blocked and/or content being hidden, which is measured by feedback ($\widehat{C}_A$, $\widehat{C}_I$, $\widehat{C}_T$) defined in Eq. (4.1). We design a reward function $\mathcal{R}_F:\mathbb{R}^3\rightarrow[-1,1]$ that mimics the FL author's assessment (Task 3) of whether a filter rule F is

good ($\mathcal{R}_F(w,\widehat{C}_A,\mathcal{B})>0$) or bad ($\mathcal{R}_F(w,\widehat{C}_A,\mathcal{B})<0$) at blocking ads based on the site feedback:

$$\mathcal{R}_F(w,\widehat{C}_A,\mathcal{B})=\begin{cases} -1 & \text{if } \widehat{C}_A=0 \quad (4.3a) \\ 0 & \text{if } \widehat{C}_A>0 \text{ , } 1-\mathcal{B}<w \quad (4.3b) \\ \widehat{C}_A & \text{if } \widehat{C}_A>0 \text{ , } 1-\mathcal{B}\geq w \quad (4.3c) \end{cases}$$

The rationale for this design is as follows.

a) *Bad Rules* (Eq. (4.3a)): If the action does not block any ads ($\widehat{C}_A=0$), the agent receives a reward value of -1 to denote that this is not a useful rule to consider.

b) *Potentially Good Rules* (Eq. (4.3b)): If the rule blocks some ads ($\widehat{C}_A > 0$) but incurs breakage beyond the FL author's tolerable breakage, then it is considered as "potentially good"[1] and receives a reward value of zero.

c) *Good Rules* (Eq. (4.3c)): If the rule blocks ads[2] and causes no more breakage than what is tolerable for the FL author, then the agent receives a positive reward based on the fraction of ads that it blocked ($\widehat{C}_A$).

4.3.2.3 Policy

Our goal is to identify "good" filter rules, *i.e.,* rules that give consistently high rewards. To that end, we need to refine our notion of a "good" rule and define a strategy for exploring the space of candidate filter rules.

Expected Reward $\mathbf{Q_t(a)}$. The MAB agent selects an action a, following a policy, from a set of available actions $\mathcal{A}$, and applies it on the site to receive a reward ($r_t=\mathcal{R}_F(w,\widehat{C}_A,\mathcal{B})$). It does this over some time horizon $t=1,\ 2,\ ...,\ T$. However, due to the site dynamics as explained in Task 4, the reward varies over time, and we need a different metric that captures

[1] "Potentially" means that the rule may have children rules within the action space that are effective at blocking ads with less breakage.

[2] Eq. (4.3) explicitly requires a rule to block at least some ads, to receive a positive reward. AutoFR can select rules that have additional side-benefits (*e.g.,* also blocks tracking requests, typically related to ads).

how good a rule is over time. In MAB, this metric is the weighted moving average of the rewards over time: $Q_{t+1}(a) = Q_t(a) + \alpha(r_t - Q_t(a))$, where α is the learning step size.

Policy. Due to the large scale of the problem and the cost of exploring candidate rules, the agent should spend more time exploring good actions. The MAB policy utilizes $Q_t(a)$ to balance between exploring new rules in $\mathcal{A}_H$ and exploiting the best known a so far. This process automates Task 1 and 2.

We use a standard Upper Bound Confidence (UCB) policy to manage the trade-off between exploration and exploitation [22]. Instead of the agent solely picking the maximum $Q_t(a)$ at each t to maximize the total reward, UCB considers an exploration value $U_t(a)$ that measures the confidence level of the current estimates, $Q_t(a)$. An MAB agent that follows the UCB policy selects a at time t, such that $a_t = \text{argmax}_a[Q_t(a) + U_t(a)]$. Higher values of $U_t(a)$ mean that a should be explored more. It is updated using $U_t(a) = c \times \sqrt{\frac{\log N[a']}{N[a]}}$, where $N[a']$ is the number of times the agent selected all actions (a') and $N[a]$ is the number of times the agent has selected a, and c is a hyper-parameter that controls the amount of exploration.

4.3.3 The AutoFR Algorithm

Algorithm 1 summarizes our AutoFR algorithm. The inputs are the site ℓ that we want to create filter rules for, the design parameter (threshold) w, and various hyper-parameters, discussed in Sec. 4.5.1.1. In the end, it outputs a set of filter rules $\mathcal{F}$, if any. It consists of the two procedures discussed next.

Initialize Procedure. First, we obtain the baseline representation of a site of interest ℓ (Sec. 4.3.2.2), when no filter rules are applied. It will visit the site n times (*i.e.,* VisitSite) to capture some dynamics of ℓ. The environment will return the average counters $\overline{C}_A$,$\overline{C}_I$,$\overline{C}_T$, and the outgoing *reqs*. The average counters are used by the reward function (Eq. (4.3)). Next, we build the hierarchical action space $\mathcal{A}_H$ using all network requests *reqs* (Task 1, 2).

AutoFR Procedure. This is the core of AutoFR algorithm. We call INITIALIZE and then traverse the action space $\mathcal{A}_H$ from the root node to get the first set of arms to consider, denoted as $\mathcal{A}$. Note that we treat every layer ($\mathcal{A}$) of $\mathcal{A}_H$ as a separate *run* of MAB with independent arms (filter rules).

One run of MAB starts by initializing the expected values of all "arms" at Q_0 and then running UCB for a time horizon T, as explained in Sec. 4.3.2.3. Since the size of $\mathcal{A}$ can change at each run, we scale T based on the number of arms; by default, we used $100 \times \mathcal{A}.size$. Each run of the MAB ends by checking the candidates for filter rules. In particular, we check if a filter rule should be further explored (down the $\mathcal{A}_H$) or become part of the output set $\mathcal{F}$, using Eq. (4.3) as a guide. A technicality is that Eq. (4.3b) compares the reward $\mathcal{R}_F$ to zero, while in practice, $Q(a)$ may not converge to exactly zero. Therefore, we use a noise threshold ($\epsilon = 0.05$) to decide if $Q_t(a)$ is close enough to zero ($-\epsilon \leq Q(a) \leq \epsilon$). Then, we apply the same intuition as in Eq. (4.3) but using $Q(a)$, instead of R_F, to assess the rule and next steps.

a) *Bad Rules: Ignore.* This case is not explicitly shown but mirrors Eq. (4.3a). If a rule is $Q(a) < \epsilon$, then we ignore it and do not explore its children.
b) *Potentially Good Rules: Explore Further.* Mirroring Eq. (4.3b), if a rule is within a range of $\pm\ \epsilon$ of zero, it helps with blocking ads but also causes more breakage than it is acceptable (w). In that case, we ignore the rule but further explore its children within $\mathcal{A}_H$. An example based on *doubleclick.net* is shown in Fig. 4.2(a). In that case, $\mathcal{A}$ is reset to be the immediate children of these arms, and we proceed to the next MAB run.
c) *Good Rules: Select.* When we find a good rule ($Q(a) > \epsilon$), we add that rule to our list $\mathcal{F}$ and no longer explore its children. This mimics Eq. (4.3c). An example is shown in Fig. 4.2(a): if *doubleclick.net* is a good rule, then its children are not explored further.

We repeatedly run MAB until there are no more potentially good filter rules to explore[3].

[3]When we find a rule that we cannot apply, we put it to "sleep", in MAB terminology. This is because they

Algorithm 1 AutoFR Algorithm

Require:

Design-parameter: $w \in [0,1]$

Inputs: Site (ℓ)
Reward function ($\mathcal{R}_F : \mathbb{R}^3 \rightarrow [-1,1]$)
Noise threshold ($\epsilon = 0.05$)
Number of site visits ($n = 10$)

Hyper-parameters: Exploration for UCB ($c = 1.4$)
Initial Q-value ($Q_0 = 0.2$)
Learning step size ($\alpha = \frac{1}{N[a]}$)
Time Horizon (T)

Output: Set of filter rules ($\mathcal{F}$)

1:
2: **procedure** INITIALIZE(ℓ, n)
3: $\overline{C}_A, \overline{C}_I, \overline{C}_T, reqs \leftarrow$ VISITSITE(ℓ, n, $\emptyset$)
4: $\mathcal{A}_H \leftarrow$ BUILDACTIONSPACE($reqs$)
5: **return** $\overline{C}_A, \overline{C}_I, \overline{C}_T, \mathcal{A}_H$
6: **end procedure**
7:
8: **procedure** AUTOFR(ℓ, w, c, α, n)
9: $\overline{C}_A, \overline{C}_I, \overline{C}_T, \mathcal{A}_H \leftarrow$ INITIALIZE(ℓ,n)
10: $\mathcal{F} \leftarrow \emptyset$, $\mathcal{A} \leftarrow \emptyset$
11: $\mathcal{A} \leftarrow \mathcal{A}_H$.root.children
12: **repeat**
13: $Q(a) \leftarrow Q_0, \forall a \in \mathcal{A}$
14: **for** $t = 1$ to T **do**
15: $a_t \leftarrow$ CHOOSEARMUCB($\mathcal{A}$, Q_t, c)
16: $C_{A_t}, C_{I_t}, C_{T_t}$, $hits \leftarrow$ VISITSITE(ℓ, 1, a_t)
17: $\widehat{C}_{A_t}, \widehat{C}_{I_t}, \widehat{C}_{T_t} \leftarrow$ SITEFEEDBACK($C_{A_t}, C_{I_t}, C_{T_t}$)
18: $\mathcal{B}_t \leftarrow$ BREAKAGE($\widehat{C}_{I_t}, \widehat{C}_{T_t}$)
19: **if** $a_t \in hits$ **then**
20: $r_t \leftarrow \mathcal{R}_F(w, \widehat{C}_{A_t}, \mathcal{B}_t)$
21: $Q_{t+1}(a_t) \leftarrow Q_t(a_t) + \alpha(r_t - Q_t(a_t))$
22: **else**
23: Put a_t to sleep
24: **end if**
25: **end for**
26: $\mathcal{A} \leftarrow \{a.\text{children}, \forall a \in \mathcal{A} \mid -\epsilon <= Q(a) <= \epsilon\}$
27: $\mathcal{F} \leftarrow \mathcal{F} \cup \{\forall a \in \mathcal{A} \mid Q(a) > \epsilon\}$
28: **until** $\mathcal{A}$ is $\emptyset$
29: **return** $\mathcal{F}$
30: **end procedure**

This stopping condition automates Task 5. The output is the final set of good filter rules $\mathcal{F}$.

do not block any network request (*i.e.,* no hits, in Task 3), and we expect them to not likely affect the site in the future, either.

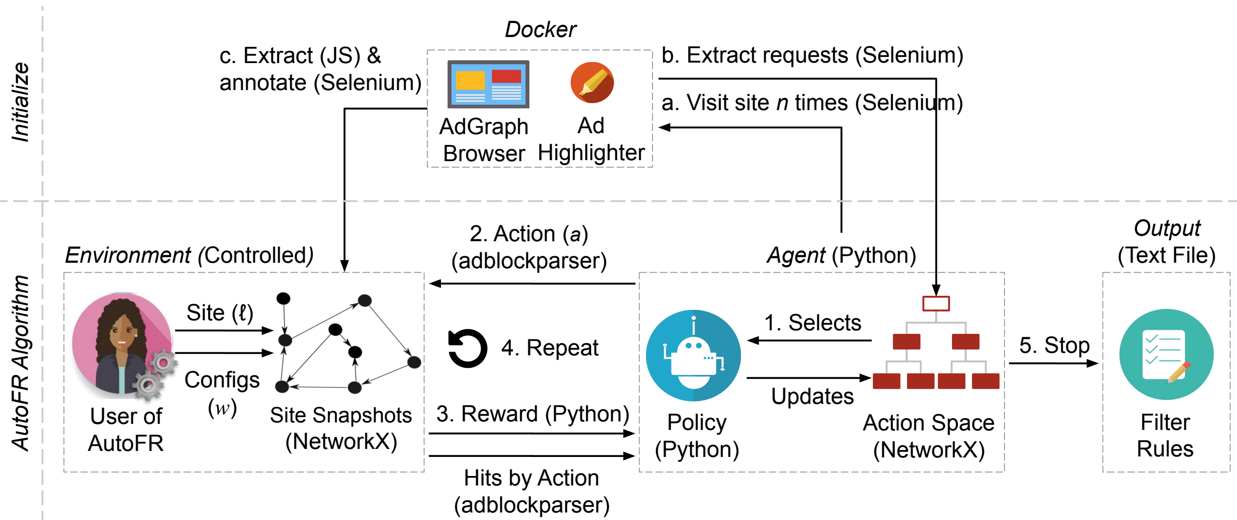


Figure 4.3: **AutoFR Example Workflow (Controlled Environment).** INITIALIZE (a–c): (a) spawns $n=10$ docker instances and visits the site until it finishes loading; (b) extracts the outgoing requests from all visits and builds the action space; (c) extracts the raw graphs and annotates them to denote C_A, C_I, and C_T, using JS and Selenium. Once all 10 snapshots are annotated, we run the RL portion of the AUTOFR procedure (steps 1–4). Lastly, AutoFR outputs the rules at step 5, *e.g.,* ||*s.yimg.com/rq/darla/4-10-0/html/r-sf.html*.

4.4 AutoFR Implementation

In this section, we present the AutoFR tool that fully implements the RL framework as described in the previous section. AutoFR removes the human-in-the-loop. The FL author only needs to provide their preferences (*i.e.,* how much they care about avoiding breakage via w) and hyper-parameters (detailed in Alg. 1), and the site of interest ℓ. AutoFR then automates Tasks 1– 5 and outputs a list of filter rules $\mathcal{F}$ specific to ℓ, and their corresponding values Q.

Implementation Costs. Let us revisit Fig. 4.1(b) and reflect on the interactions with the site. The MAB agent (as well as the human FL author) must visit the site ℓ, apply the filter rule, and wait for the site to finish loading the page content and ads (if any). The agent must repeat this several times to learn the expected reward of rules in the set of available actions $\mathcal{A}$. First, for completeness, we implemented exactly that in a live environment, referred to as AutoFR-L with details in Sec. 4.7.

We employed cloud services using Amazon Web Services (AWS) to scale to tens of thousands of sites. This has high computation and network access costs and, more importantly, introduces long delays until convergence.

To make things concrete. For the delay, we found it took 47 seconds per-visit to a site, on average, by sampling 100 sites in the Top–5K. Thus, running AutoFR for one site with ten arms in the first MAB run, for 1K iterations, would take 13 hours for one site alone! For the monetary cost, running AutoFR-L on 1K sites and scaling it using one AWS EC2 instance per-site ($0.10/hour) would cost roughly $1.3K for 1K sites, or $1.3 to run it once per-site. This a well-known problem with applying RL in a real-world setting. Thus, an implementation of AutoFR that creates rules by interacting with live sites is inherently slow, expensive, and does not scale to a large number of sites.

Scalable and Practical. Although AutoFR-L is already an improvement over the human workflow, we were able to design an even faster tool, which produces rules for a single site in minutes instead of hours. The core idea is to create rules in a realistic but controlled environment, where the expensive and slow visits to the website are performed in advance, stored once, and then used during multiple MAB runs, as explained in Sec. 4.3.3. In this section, we present the design of this implementation in a controlled environment: AutoFR-C, or AutoFR for simplicity. An implementation overview is provided in Fig. 4.3. Importantly, this allows our AutoFR tool to scale across thousands of sites and, thus, utilized as a practical tool.

4.4.1 Environment

To deal with the aforementioned delays and costs during training, we replace *visiting* a site live with *emulating* a visit to the site, using saved site snapshots. This provides advantages: (1) we can parallelize and speed up the collection of snapshots, and then run MAB off-line; (2) we can reuse the same stored snapshots to evaluate different w values, algorithms, or

reward functions while incurring the collection cost only once; and (3) we plan to make these snapshots available to the community.

Collecting and Storing Snapshots. Site snapshots are collected up-front during the INITIALIZE phase of Alg. 1 and saved locally. We illustrate this in Fig. 4.3, steps a–c. We use AdGraph [73], an instrumented Chromium browser that outputs a graph representation of how the site is loaded. To capture the dynamics, we visit a site multiple times using Selenium to control AdGraph and collect and store the site snapshots. The environment is dockerized using Debian Buster as the base image, making the setup simple and scalable. For example, we can retrieve 10 site snapshots in parallel, if the host machine can handle it. In Sec. 4.5.1, we find that a site snapshot takes 49 seconds on average to collect. Without parallelization, this would take 8 minutes to collect 10 snapshots sequentially.

Defining Site Snapshots. Site snapshots represent how a site ℓ is loaded. They are directed graphs with known root nodes and possible cycles. An example is shown in Fig. 4.4. Site snapshots are large and contain thousands of nodes and edges, shown later in Fig. 4.6. We use AdGraph as the starting point for defining the graph structure and build upon it. First, we automatically identify the visible elements, *i.e.,* ads (AD), images (IMG), and text (TEXT) (technical details in Sec. 4.4.3), for which we need to compute counts C_A, C_I, and C_T, respectively. Second, once we identify them, we make sure that AdGraph knows that these elements are of interest to us. Thus, we annotate the elements with a new attribute such as "FRG-ad", "FRG-image", and "FRG-textnode" set to "True". Annotating is challenging because ads have complex nested structures, and we cannot attach attributes to text nodes. Third, we include how JS scripts interact with each other using "Script-used-by" edges, shown in Fig. 4.4. Lastly, we save site snapshots as ".graphml" files.

Emulating a Visit to a Site. Emulation means that the agent does not actually visit the site live but instead reads a site snapshot and traverses the graph to infer how the site was

loaded. To emulate a visit to the site, we randomly read a site snapshot into memory using NetworkX and traverse the graph in a breadth-first search manner starting from the root — effectively replaying the events (JS execution, HTML node creation, requests that were initiated, etc.) that happened during the loading of a site. This greatly increases the performance of AutoFR as the agent does not wait for the per-site visit to finish loading or for ads to finish being served. Thus, reducing the network usage cost. We hard-code a random seed (40) so that experiments can be replicated later.

Applying Filter Rules. To apply a filter rule, we use an offline adblocker, adblockparser [120], which can be instantiated with our filter rule. If a site snapshot node has a URL, we can determine whether it is blocked by passing it to adblockparser. We further modified adblockparser to expose which filter rules caused the blocking of the node (*i.e.,* hits). If a node is blocked, we do not consider its children during the traversal.

Capturing Site Feedback from Site Snapshots. The next step is to assess the effect of applying the rule on the site snapshot. At this point, the nodes of site snapshots are already annotated. We need to compute the counters of ads, images, and text (C_A, C_I, C_T), which are then used to calculate the reward function. Its Python implementation follows Sec. 4.3.2.2.

We use the following intuition. If we block the source node of edge types "Actor", "Requestor", or "Script-used-by", then their annotated descendants (IMG, TEXT, AD) will be blocked (*e.g.,* not visible or no longer served) as well. Consider the following examples on Fig. 4.4: (1) if we block JS Script A, then we can infer that the annotated IMG and TEXT will be blocked; (2) if we block the annotated IMG node itself, then it will block the URL (*i.e.,* stop the initiation of the network request), resulting in the IMG not being displayed; and (3) if we block JS Script B that is used by JS Script A, then the annotated nodes IMG, TEXT, IFRAME (AD) will all be blocked. As we traverse the site snapshot, we count as follows. If we encounter an annotated node, we increment the respective counters C_A. C_I,

C_T. If an ancestor of an annotated node is blocked, then we do not count it.

Limitations. To capture the site dynamics due to a site serving different content and ads, we perform several visits per-site and collect the corresponding snapshots. We found that 10 visits were sufficient to capture site dynamics in terms of the eSLDs on the site, which is a similar approach taken by prior work [85, 149]. We describe why 10 visits are enough in Sec. 4.5.1.1. However, there is also a different type of dynamics that snapshots miss. When we emulate a visit to the site while applying a filter rule, we infer the response based on the stored snapshot. In the live setting, the site might detect the adblocker (or detect missing ads [85]) and try to evade it (*i.e.,* trigger different JS code), thus leading to a different response that is not captured by our snapshots. We evaluate this limitation in Sec. 4.5.1.3 and show that it does not impact the effectiveness of our rules. Another limitation can be explained via Fig. 4.4. When JS Script B is used by JS Script A, we assume that blocking B will negatively affect A. Therefore, if A is responsible for IMG and TEXT, then blocking B will also block this content; this may not happen in the real world. When we did not consider this scenario, we found that AutoFR may create filter rules that cause major breakage. Since breakage must be avoided and we cannot differentiate between the two possibilities, we maintain our conservative approach.

4.4.2 Agent

Action Space $\mathcal{A}_H$. During the Initialize procedure (Alg. 1), we visit the site ℓ multiple times and construct the action space. First, we convert every request to three different filter rules, as shown in Table 4.1. We add edges between them (eSLD $\rightarrow$ FQDN $\rightarrow$ With path), which serve as the finer-grain edges, shown in Fig. 4.2(a). We further augment $\mathcal{A}_H$ by considering the "initiator" of each request, retrieved from the Chrome DevTools protocol and depicted in solid lines in Fig. 4.2(a). This makes the $\mathcal{A}_H$ taller and reduces the number of arms to explore per run of MAB, as described in Sec. 4.3.3. The resulting action space is a directed acyclic graph with nodes that represent filter rules. Fig. 4.2(a) provides a zoom-in

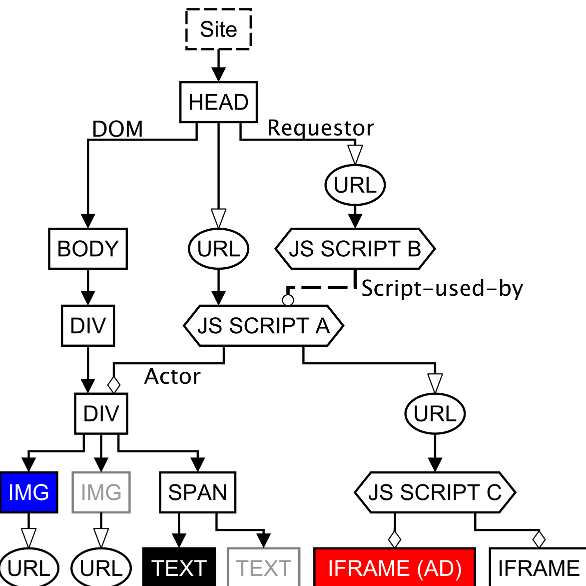


Figure 4.4: **Site Snapshot.** It is a graph that represents how a site is loaded. The nodes represent JS Scripts, HTML nodes (*e.g.,* DIV, IMG, TEXT, IFRAME), and network requests (*e.g.,* URL). "Actor" edges track which source node added or modified a target node. "Requestor" edges denote which nodes initiated a network request. "DOM" edges capture the HTML structure between HTML nodes. Lastly, "Script-used-by" edges track how JS scripts call each other. As described in Sec. 4.4.1, nodes annotated by AutoFR have filled backgrounds, while grayed-out nodes are invisible to the user.

example. We implement it as a NetworkX graph and save it as a ".graphml" file, a standard graph file type utilized by prior work [124].

Policy. The UCB policy of Sec. 4.3.2.3 is implemented in Python. At time t (Alg. 1, line 14), the agent retrieves the filter rule selected by the policy and applies it on the randomly chosen site snapshot instance.

4.4.3 Automating Visual Component Detection

A particularly time-consuming step in the human workflow is Task 3 in Fig. 4.1(a). The FL author visually inspects the page, before and after they apply a filter rule, to assess whether the rule blocked ads ($\widehat{C}_A$) and/or impacted the page content ($\widehat{C}_I$, $\widehat{C}_T$). AutoFR in Fig. 4.1(b) summarizes this assessment in the reward in Eq. (4.3). However, to minimize the human

work, we also need to replace the visual inspection and automatically detect and annotate elements as ads (AD), images (IMG), or text (TEXT) on the page.

Detection of AD (Perceptual). To that end, we automatically detect ads using Ad Highlighter [127], a perceptual ad identifier (and web extension) that detects ads on a site. We evaluated different ad perceptual classifiers, including Percival [2], and we chose Ad Highlighter because it has high precision and does *not* rely on existing filter rules. We utilize Selenium to traverse nested iframes to determine whether Ad Highlighter has marked them as ads. The details of how Ad Highlighter works are deferred to Sec. 4.7.2.1.

Detection of IMG and TEXT. We automatically detect visible images and text by using Selenium to inject our custom JS that walks the HTML DOM and finds image-related elements (*i.e.,* ones that have background-urls) or the ones with text node type, respectively. To know if they are visible, we see whether the element's or text container's size is $>$ 2px [85].

Discussion of the Visual Components. It is important to note that our framework is agnostic to how we detect elements on the page. For detecting ads, this can be done by a human, the current Ad Highlighter, future improved perceptual classifiers, heuristics, or any component that identifies ads with high precision. This also applies to detecting the number of images and text. Images can be counted using an instrumented browser that hooks into the pipeline of rendering images [2]. Text can be extracted from screenshots of a site using Tesseract [127], an OCR engine. Therefore, the AutoFR framework is modular and dependent on how well these components perform.

Discussion of Blocking Ads vs. Tracking. We focus on detecting ads and generating filter rules that block ads for two reasons. First, they are the most popular type of rules in filter lists. Second, ads can be visually detected, enabling a human (FL author) or a visual detection module (such as Ad Highlighter) to assess if the rule was successful (the ad is no longer displayed) or not at blocking ads. Although tracking is related to ads, it is impossible to de-

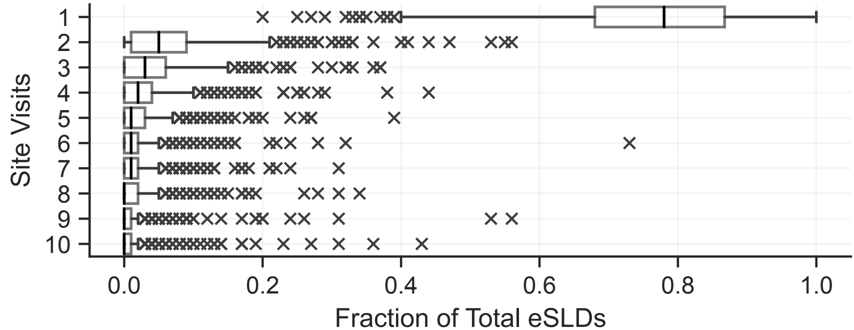


Figure 4.5: **Site Dynamics.** We consider site dynamics as the unique eSLDs a site contacts. Using our *Full-W09-Dataset*, we show the fraction of unique eSLDs that we collect after every subsequent visit. By the fifth visit, we collected the majority of site dynamics for most sites within our dataset. Besides some outliers, visiting a site 10 times is more than enough to capture site dynamics.

tect visually, and assessing the success of a rule that blocks tracking is more challenging, *e.g.,* involves JS code analysis [43]. Extending AutoFR for tracking is a direction for future use.

4.5 Evaluation

In this section, we evaluate the performance of AutoFR (*i.e.,* the trade-off between blocking ads and avoiding breakage) and compare it to EasyList as a baseline. We characterize properties of the filter rules produced by AutoFR: how they can be controlled via parameter w, how they compare to EasyList rules, and how fast they need to be updated.

4.5.1 Filter Rule Evaluation Per-Site

4.5.1.1 Parameter Selection for AutoFR

We select $w = 0.9$ to represent a user who has similar interests to FL authors but has a slightly higher tolerance for breakage. This user wants filter rules that block ads with minimal breakage. We further explore how changing w affects the output of AutoFR in Sec. 4.5.1.5. As explained in Sec. 4.3.3, we have several hyper-parameters that need to be tuned. We list the

choices of these hyper-parameters as follows:

- *Initial estimates (Q_0)*: We use the optimistic initial value approach for MAB [128]. Every filter rule may block ads if the MAB selects it. However, we do not want to go too far above zero, as the rules that are "potentially good" need to converge near zero (see Sec. 4.3.3). Hence, we chose $Q_0 = 0.2$ as an initial value. This allows every filter rule to be tested by the MAB agent.
- *Learning rate (α)*: We use an adaptive learning rate $\alpha = \frac{1}{N[a]}$ to update the $Q(a)$ values. Here, $N[a]$ is the number of times the action a has been selected. We adopted this approach over a constant learning rate to capture the fact that rules can vary in their effectiveness. For example, on one extreme, if the rule *googlesyndication.com* gets $r = 1$ for the first 10 pulls by the MAB but then does not work at all on the $11th$ pull ($r = -1$), then $Q(a)$ would be dramatically affected with a high constant learning step.
- *Exploration rate for UCB (c):* We set the exploration rate for UCB $c = 1.4$, to encourage AutoFR to explore the arms without prolonging the convergence of the algorithm greatly (*e.g.,* $c = 2$ causes the convergence to take twice as long).
- *Site Dynamics:* To capture site dynamics, we visit each site multiple times. Motivated by prior work, we determine the number of visits necessary by counting the number of new eSLDs captured after every visit incrementally. Fig. 4.5 reveals that 10 visits are more than enough to capture the site dynamics. This more than doubles prior work [85, 149].

4.5.1.2 Experiment Setup

For Sec. 4.5 and 4.6, we utilize an automated approach to evaluate the effectiveness of filter rules. We discuss its limitations and confirm our results with independent visual inspections in Sec. 4.5.1.3.

- *In the Wild:* In the live setting, we apply rules to a site (for real) 10 times and capture the

Datasets $w = 0.9$	**Sites**	**Filter Rules**	**Snapshots**
W09-Dataset (Sites $\geq$ 1 rule)	933	361	9.3K
Full-W09-Dataset (All sites)	1042	361	10.4K

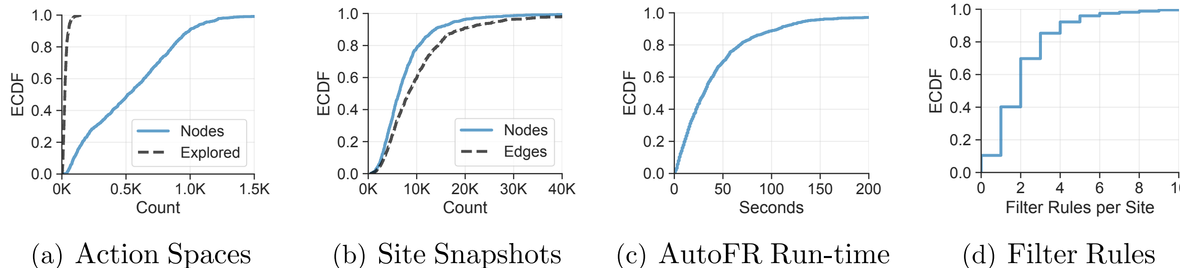
Table 4.2: **AutoFR Top–5K Datasets**

site feedback C_A, C_I, C_T, as described in Sec. 4.4.1. We then average the values and use that to calculate our trade-off terms of blocking ads $\widehat{C_A}$ (Eq. (4.1)) and avoiding breakage $1-\mathcal{B}$ (Eq. (4.2)). We use the Tranco list [86, 131]. Since we run our experiments in the US region, we customize the list with popular sites for the US only. Next, we set up AutoFR using Amazon's Web Services (AWS) and EC2. Specifically, we use EC2 instance type *m5.2xlarge* with eight vCPU, 32GB memory, and 35GB storage. For each site, we make sure that Ad Highlighter can detect at least one ad before applying AutoFR. Specifically, we visit each site three times and consider sites with an average number of ads >0.

- *Controlled (Site Snapshots):* In a controlled environment, we apply rules to each set of 10 site snapshots for a given site to emulate its visits, as explained in Sec. 4.4.1. We proceed with the same calculations for the trade-off terms.
- *Comparing to EasyList.* EasyList is a state-of-the-art filter list for adblocking on the web [54]. However, it contains rules beyond URL-based filter rules, such as element hiding rules. Thus, for any experiment that involves EasyList, we make it comparable to our URL-based filter rules. We parse the list and utilize delimiters (*e.g.,* "$", "||", and "^") to identify URL-based filter rules and keep them.

4.5.1.3 AutoFR Results

We apply AutoFR on the Tranco Top–5K sites [86, 131] to generate rules using the breakage tolerance threshold of $w = 0.9$. All other AutoFR parameters are the same as in Alg. 1. Table 4.2 summarizes our datasets and filter rules generated on the Top–5K, while Fig. 4.6 utilizes the *Full-W09-Dataset* to characterize the sizes of action spaces and site snapshots,



(a) Action Spaces (b) Site Snapshots (c) AutoFR Run-time (d) Filter Rules

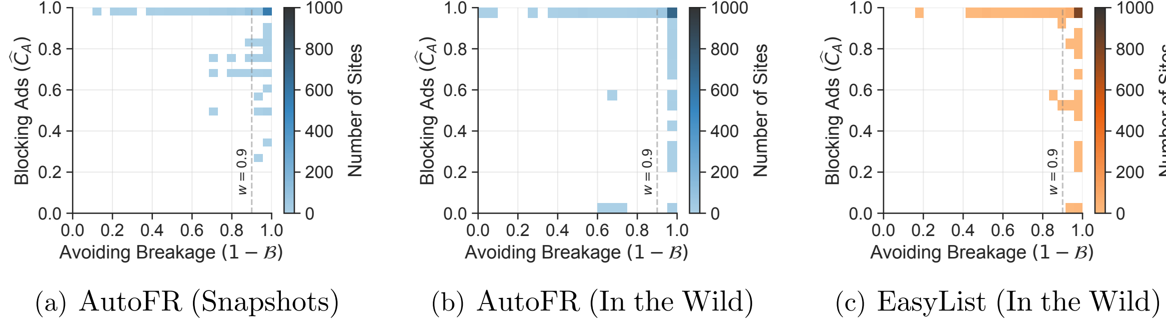
Figure 4.6: ***Full-W09-Dataset.***: *(a) Action Spaces:* 75% of action graphs have 800 nodes or fewer. AutoFR only needs to explore a fraction of the action space to find effective rules. *(b) Site Snapshots:* 75% of site snapshots contain 10K nodes or fewer. *(c) AutoFR Run–time:* 75% of sites take a minute or less to execute the multi-arm bandit portion of Alg. 1. *(d) Filter Rules:* For 75% of sites, AutoFR generated three filter rules or fewer.

		Sec. 4.5.1, Fig. 4.7, Top–5K				Sec. 4.5.3.1
		AutoFR (Snapshots) (Jan. 2022)	**AutoFR** (In-the-Wild) (Jan. 2022)	**AutoFR** (*Confirm) (In the Wild)	**EasyList** (In the Wild) (Jan. 2022)	**AutoFR** (In the Wild) (July 2022)
	Description ($w = 0.9$)	1	2	3	4	5
1	Sites in operating point: $\widehat{C}_A \geq 0.95$, $1 - \mathcal{B} \geq 0.95$	62%	74%	85%	79%	72%
2	Sites within w: $\widehat{C}_A > 0$, $1 - \mathcal{B} \geq 0.9$	77%	86%	85%	87%	82%
3	Ads blocked within w: $\sum_\ell (\overline{C}_A \times \widehat{C}_A) \,/\, \sum_\ell \overline{C}_A$; $1 - \mathcal{B} \geq 0.9$	70%	86%	84%	87%	78%

Table 4.3: **Results.** We provide additional results to Fig. 4.7. We explain the meaning of each row: (1) the number of sites that are in the operating point (top-right corner of the figures), where filter rules were able to block the majority of ads with minimal breakage; (2) the number of sites that are within w; and (3) the fraction of ads that were blocked across all ads within w. **Confirming via Visual Inspection (In the Wild)* (Sec. 4.5.1): col. 3 is based on a binary evaluation. As it is not simple for a human to count the exact number of missing images and text, we evaluate each site based on whether the rules blocked all ads or not (*i.e.*, $\widehat{C}_A$ is either 0 or 1) and whether they caused breakage or not (*i.e.*, $\mathcal{B}$ is either 0 or 1). For col. 5 (Sec. 4.5.3.1), we repeat the same experiment of col. 2 during July 2022 for a longitudinal study of AutoFR rules.

the run-time of AutoFR and the number of rules generated per-site. Overall, AutoFR generated 361 filter rules for 933 sites. For some sites, AutoFR did not generate any rules since none of the potential rules were viable at the selected w threshold.

Efficiency. AutoFR is efficient and practical: it can take 1.6–9 minutes to run per-site,



(a) AutoFR (Snapshots) (b) AutoFR (In the Wild) (c) EasyList (In the Wild)

Figure 4.7: **AutoFR (Top–5K).** All sub-figures exhibit similar patterns. First, the filter rules were able to block ads with minimal breakage for the majority of sites. Thus, the top-right bin (the operating point) is the darkest. Second, there are edge cases for sites with partially blocked ads within the w threshold (right of w line) and sites below the w threshold (left of w line). See Table 4.3, col. 1, 2, and 4, for additional information.

which is an order of magnitude improvement over the 13 hours per-site of live training in Sec. 4.4. During each per-site run, we explore tens to hundreds of potential rules and conduct up to thousands of iterations within MAB runs (see Fig. 4.6). This efficiency is key to scaling AutoFR to a large number of sites and over time.

AutoFR: Validation with Snapshots. Since AutoFR generates rules for each particular site (*i.e.,* per-site), we first apply these rules to the site for which they have been created. To that end, we first apply the rules to the stored site snapshots, and we report the results in Fig. 4.7(a) and Table 4.3 col. 1. We see that the rules block ads on 77% of the sites within the $w = 0.9$ breakage threshold. As we demonstrate next, this number is lower due to the limitations of traversing snapshots (Sec. 4.4.1) and the rules are more effective when tested on sites in the wild.

AutoFR *vs.* EasyList: Validation In The Wild. EasyList[4] to the same set of Top–5K sites and we report our results in Fig. 4.7(b) and Table 4.3 col. 2 and 4. AutoFR's rules block 95% (or more) of ads with less than 5% breakage for 74% of the site (*i.e.,* within the operating point) as compared to 79% for EasyList. For sites within the w threshold, AutoFR

[4]For a fair comparison, we parse EasyList and utilize delimiters (*e.g.,* "\$", "||", and "^") to identify URL-based filter rules and keep them.

and EasyList perform comparably at 86% and 87%, respectively (row 2). Overall, our rules blocked 86% of all ads *vs.* 87% by EasyList, within the w threshold (row 3). Some sites fall below the w threshold partly due to the limitations of AdGraph [73].

To further confirm our results for AutoFR and EasyList, we randomly selected 272 sites (a sample size out of 933 sites to get a confidence level of 95% with a 5% confidence interval), and we visually inspected them. In particular, we looked for breakage not perfectly captured by automated evaluation. Table 4.3 col. 3 summarizes the results and confirms our results obtained through the automated workflow. We find that 3% (7/272) of sites had previously undetected breakage. For instance, the layout of four sites was broken (although all of the content was still visible), and one site's scroll functionality was broken. Note that this kind of functionality breakage is currently not considered by AutoFR. We find three sites had some images missing. This was because images were served with "<amp-img>" instead of the standard "<img>" tag. This can be easily addressed by updating how we retrieve C_A in Sec. 4.4.3. We observed two sites (*e.g., gazeta.ru*) that intentionally caused breakage (the site loads the content, then goes blank) after detecting their ads were blocked. AutoFR's implementation currently does not handle this type of adblocking circumvention. AutoFR can generate filter rules that block ads that do not have ad transparency logos. We observe that our filter rules could block all ads for 90% (44/49) of sites that also served non-transparency ads. We deduce this is because a site will use the same approach (or JS) to serve ads with and without ad transparency logos.

4.5.1.4 Validation of Detection Modules

In this section, we validate the detection of AD, IMG, and TEXT used in AutoFR.

Detection of AD (Perceptual): Validation In The Wild. We leverage the same sample of 272 sites to keep track of ads detected by Ad Highlighter to validate its precision. We count a total of 1040 ads that were detected by Ad Highlighter. We found five false positives

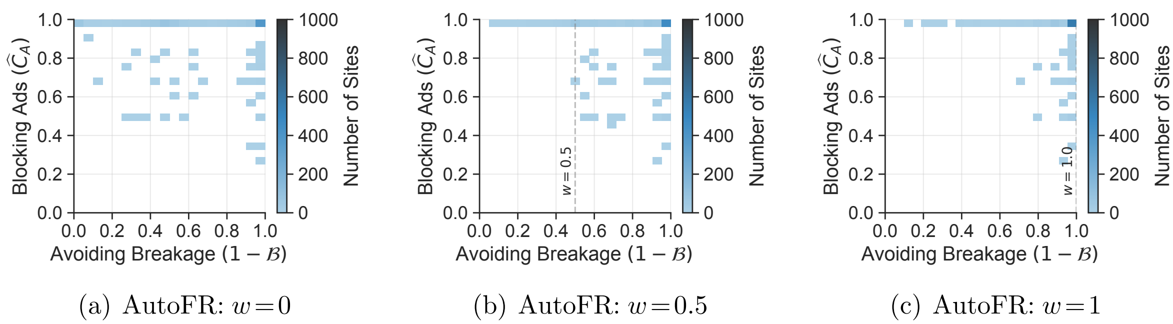
(*i.e.,* not ads), giving us a 99% precision in ads. When we consider it in terms of sites, this affected 2% of sites (5/272). False positives can appear due to social widgets like Twitter and SoundCloud with play buttons similar to AdChoice logos. However, we note that this does not always happen for every embedded social widget.

Detection of IMG and TEXT: Validation In The Wild. To validate our methodology of capturing the number of visible images (C_I) and text (C_T) for a given site, we randomly sample 100 sites from *W09-Dataset* and modify our custom JS in the following ways. For images that we identify, we add a blue solid border; for text, we append "(AutoFR)". For each site, we automatically visit the site using Selenium and inject the modified JS before taking a screenshot. We then visually inspected the 100 screenshots to see whether the images and text were correctly captured. Note that the images and text captured must not be part of ads. We observe our methodology has 100% precision in capturing visible images and text. This is not surprising, as our methodology relies on common approaches to display images (using *img* tags, and "background-url") and text (we only consider HTML nodes with the type "TEXT_NODE" [91]).

Next, we evaluate the images and text that were missed. First, we utilize the screenshots to find the locations of visible content that were missed and keep track of their counts. Then, we visit the site manually using a Chrome browser, inject the JS using the Chrome Developer Tools, and inspect the HTML DOM to discern the reason for the missed content. For instance, we miss some visible images and text because they are rendered using <*svg*> or pseudo-elements. However, most of this missed content is small icons for social media sharing (*e.g.,* Facebook, Twitter), top menus, and footers. If we consider the missed images as false negatives, we get a recall of 95% for capturing visible images. For capturing visible text, we get a recall of 99%. Future improvements to AutoFR can consider <svg> for C_I by modifying the JS. However, for pseudo-elements, we would need to modify the browser to capture these images. Fortunately, these are often trivial images such as small icons.

Threshold w	Filter Rules for womenshealthmag.com
0	||*doubleclick.net*ˆ, ||*googlesyndication.com*ˆ, ||*hearstapps.com*ˆ
0.1–0.5	||*doubleclick.net*ˆ, ||*hearstapps.com*ˆ
0.6–0.9	||*doubleclick.net*ˆ, ||*assets.hearstapps.com*ˆ, ||*amazon-adsystem.com*ˆ, ||*googletagmanager.com*ˆ
1	||*doubleclick.net*ˆ, ||*amazon-adsystem.com*ˆ, ||*googletagmanager.com*ˆ, ||*assets.hearstapps.com/moapt/moapt-hdm.latest.js*, ||*assets.hearstapps.com/moapt/moapt-bidder-pb.*.js*

Table 4.4: **Effects of w.** We show how w changes the generated rules for one site. As w increases, some rules will no longer be outputted and new rules may be discovered. While others will become more specific.



(a) AutoFR: $w=0$ (b) AutoFR: $w=0.5$ (c) AutoFR: $w=1$

Figure 4.8: **AutoFR across Different w (Top–5K).** We run AutoFR on *Full-W09-Dataset* using a range of $w \in [0,0.5,1]$ and visualize the effectiveness based on the trade-off of blocking ads *vs.* avoiding breakage. As w increases, there are more sites in operating point. Lower w denotes that the user does not care about breakage, which causes less exploration of the action space for rules that fall in the operating point.

4.5.1.5 Tuning AutoFR via Threshold w

AutoFR is the first approach that can be tuned per-site and explicitly allows one to express a preference. The FL author that uses AutoFR must select the site to create rules for and express their preference by tuning a knob (threshold w). There is no optimized value for threshold w, but rather it is a manually selected value by the user of AutoFR.

How does the Trade-off Change as w Increases? Fig. 4.8 illustrates the trade-off as w increases on the entire *Full-W09-Dataset* for each individual w value. First, for low w's, we notice more breakage. This is not surprising as the user does not care about breakage. As

w increases, we can see that the filter rules adhere to the threshold and mostly stay within it (*i.e.,* being on the right side of w). However, interestingly, we observe that there are more sites that are in the operating point of the plots (*i.e.,* the top-right corner). This is because as the user cares more about breakage, AutoFR is exploring more of the action space (*i.e.,* going down the hierarchy), and thus more chances of candidate rules that are in the operating point.

How do Filter Rules Change as w Increases? Next, Table 4.4 deep dives into an example of how w changes the output of AutoFR for one site. First, filter rules can go from being part of the output to no longer part of the output, as shown with the transition of ||*googlesyndication.com*^ from $w = 0$ to $w = 0.1$. Conversely, new rules may appear as w increases, as evident with ||*amazon-adsystem.com*^ between $w = 0.5$ and $w = 0.6$. Lastly, we observe that as w increases, rules will be more specific, as shown with the progression of how ||*hearstapps.com*^ changes from eSLD in $w = 0$, to FQDN in $w = 0.6$, then to a rule with a path in $w = 1$.

4.5.2 AutoFR *vs.* EasyList: Comparing Rules

We compare the rules generated per-site by AutoFR and EasyList from Sec. 4.5.1. For a fair comparison, we only consider EasyList rules that are *triggered* when visiting sites.

4.5.2.1 Rule Type Granularity

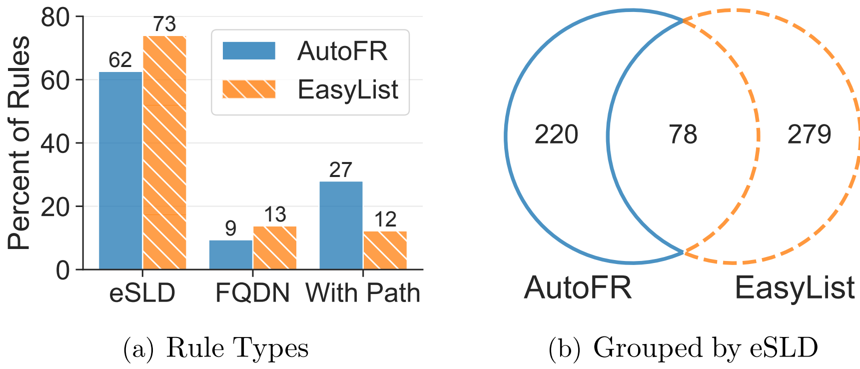
An important aspect to consider when comparing rules is the suitable granularity of the rules that block ads while limiting breakage. Fig. 4.9(a) breaks down the granularity of rules by AutoFR and EasyList. We note that both exhibit a similar distribution: eSLD rules are the most common, while the other rule types are less common. Across all granularities, there are 59 identical rules (*e.g.,* ||*pubwise.io*^, ||*adnuntius.com*^, and ||*deployads.com*^) between AutoFR and EasyList, which represents 15% of EasyList rules.

Next, we focus on rules that are *related*, *i.e.,* they share a common eSLD but may differ in subdomain or path, to understand why AutoFR generates rules that are coarser or finer-grain than EasyList rules. In Fig. 4.9(b), we show that when we group rules by eSLD, there are 78 common eSLDs, 60 (77%) of which have at least one identical rule. For example, for *mail.ru*, both AutoFR and EasyList have ||*ad.mail.ru*^.

For 26 eSLD groups, AutoFR and EasyList rules differ in granularity. First, 18 eSLDs have AutoFR rules that are coarser-grained than EasyList. For instance, AutoFR has ||*cloudfront.net*^ but EasyList has 15 variations of ||*d2na2p72vtqyok.cloudfront.net*^ based on FQDNs. CloudFront is a CDN that can serve resources for legitimate content, ads, and tracking. As AutoFR generates per-site rules, it can afford to be more coarse-grained because a site may only use CloudFront for ads and tracking. However, since EasyList rules that target CloudFront are not per-site, they are more finer-grain to avoid breakage on other sites.

Second, six eSLDs have AutoFR rules that are finer-grain than EasyList. For instance, for *moatads.com*, AutoFR has ||*z.moatads.com*^ when EasyList has ||*moatads.com*^. Recall in Sec. 4.4.1 that AutoFR generates rules with a conservative approach when using site snapshots, and will consider finer-grain rules for some cases to avoid breakage. Whereas FL authors verify rules for EasyList and will know that ||*moatads.com*^ is more appropriate.

Lastly, four eSLDs share the same granularity but contain rules that are not identical. For example, AutoFR has ||*pastemagazine.com/common/js/ads-gam-a9-ow.js*, while EasyList has *pastemagazine.com/common/js/ads-* for site *pastemagazine.com*. Partial paths within EasyList may extend the life of a filter rule over time for some sites. We further evaluate this in Sec. 4.5.3.1. AutoFR can extend to partial paths in the future.



(a) Rule Types (b) Grouped by eSLD

Figure 4.9: **Comparing AutoFR Rules to EasyList**. Some rules are common and some are unique to each approach. When comparing rules, one must consider the right granularity.

4.5.2.2 Understanding Unique Rules

We investigate why AutoFR generates rules that are not present in EasyList and vice versa. We found that when grouped by eSLD (Fig. 4.9(b)), unique rules are due to the design and implementation of our framework, as well as due to site dynamics.

Methodology. To investigate each unique rule (either from AutoFR or EasyList), we apply the rule to its corresponding site snapshots (per-site) and extract the requests that were blocked. We manually investigate these requests as follows. For images, we visually decide whether it is an ad. For scripts, we use our domain knowledge and keywords (*e.g.,* "advertising", "bid") to examine the source code to discern whether they affect ads, tracking, functionality, or legitimate site content. When we cannot determine the nature of the request (*e.g.,* due to obfuscated JS code), we fall back to applying the rule and evaluating its effectiveness via visual inspection, following the methodology in Sec. 4.5.1.

Findings. Depicted in Fig. 4.9(b), the differences in rules when grouped by eSLDs are due to three main reasons.

1. AutoFR Framework: Our framework exhibits several strengths when generating rules. 48% (105/220) of the unique eSLDs for AutoFR have rules that are valid but seem chal-

lenging for a FL author to manually craft. Within this set, 19% (20/105) are first-party (*e.g.,* *||kidshealth.org/.../inline_ad.html*), 52% (55/105) block resources that involve both ads and tracking (*e.g.,* *||snidigital.com^*), 23% (24/105) block ad-related resources served by CDNs (*e.g.,* *||cdn.fantasypros.com/realtime/media_trust.js*), and 42% (44/105) block ad-related resources served through seemingly obfuscated URLs. We conclude that AutoFR can create rules that are not obviously ad-related (*e.g.,* by looking at keywords in the URL) but are effective nonetheless.

Next, we explain how certain design decisions behind AutoFR's framework can lead to missed EasyList rules. First, AutoFR focuses on rules that block at least some ads (due to Eq. (4.3a)), which is why AutoFR ignored 10% (28/279) of unique eSLDs from EasyList that are responsible for purely tracking requests. Second, we choose to generate rules that block ads across all 10 site snapshots of a site, not just one site snapshot, to be robust against site dynamics. In addition, we choose to stop exploring the hierarchical action space when we find a good rule following the intuition from Sec. 4.3.2.1, which improves the efficiency of AutoFR. Of course, these design decisions can be altered depending on the user's preference. When we do so, we find that the overlap in Fig. 4.9(b) goes from 22% (78/357) to 35% (124/357). For example, *adtelligent.com* and *adscale.de* are new common eSLDs found when we remove these design decisions.

2. AutoFR Implementation: Our implementation of Alg. 1 focuses on visual components (*e.g.,* using Ad Highlighter to detect ads) and how filter rules affect them. The rules generated are as good as the components that we utilize. First, AutoFR misses 28% (78/279) of unique eSLDS from EasyList because Ad Highlighter can only detect ads that contain transparency logos. However, AutoFR rules are still effective when compared to EasyList, as shown in Sec. 4.5.1 and Table 4.3. This demonstrates that we do not necessarily need to replicate all rules from EasyList to be effective. Second, 18% of unique eSLDs from AutoFR can affect both ads and functionality (*e.g.,* cdn.ampproject.org/v0/amp-ad-0.1.js for ads,

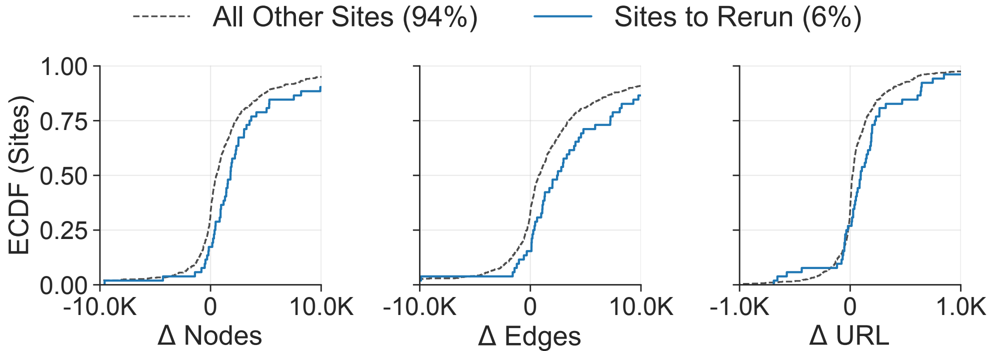


Figure 4.10: Δ **Site Snapshots between July vs. January 2022.** The differences in site snapshots for nodes, edges, and URLs. A positive change in the x-axis denotes that July had more of the respective factor, while a zero denotes no change.

amp-accordion-0.1.js for functionality). AutoFR balances the trade-off between blocked ads and breakage, see Sec. 4.5.1.

3. Site Dynamics can also lead to differences in the site resources between site snapshots *vs.* the in the wild evaluation. Due to this, 18% (50/279) of unique eSLDs on the EasyList side did not appear in our *W09-Dataset*. Thus, AutoFR did not get an opportunity to generate these rules. Conversely, 5% (11/220) of unique eSLDs from AutoFR appear in EasyList but were not triggered during the evaluation of EasyList rules. This can be mitigated by increasing the number of site snapshots used in AutoFR's rule generation or applying EasyList more times during our in the wild evaluation. Although, recall that we already do these steps for 10 times.

Takeaways. The difference in the granularity of related rules generated by AutoFR and EasyList is mainly because AutoFR creates rules per-site. Unique rules to AutoFR or EasyList are due to the design and implementation of our framework and site dynamics. These differences are acceptable because the effectiveness of the rules from AutoFR and EasyList is comparable. This is crucial from a practical standpoint.

4.5.3 Robustness of AutoFR Filter Rules

AutoFR generates rules for a particular site and uses snapshots collected at a particular time. We investigate how well these rules perform over time and in adversarial scenarios.

4.5.3.1 How Long-lived are AutoFR Rules?

Sites change naturally over time, which may result in changes in the site snapshots, and eventually into changes in the filter rules. We show that AutoFR rules remain effective for a long time and can be rerun fast when needed to update.

Efficacy of Rules Over Time. We re-apply per-site rules generated in January 2022 (Sec. 4.5.1) to the same sites in July 2022 and summarize the results in Table 4.3 (col. 5). We find that the majority of AutoFR rules are still effective after six months. 72% of sites (down only by 2%) still achieve the operating point (row 1), and 82% (down by 4%) achieve $1-\mathcal{B} \geq 0.9$ (row 2). Even more interestingly, we found only 6% of the sites now no longer have all or any ads blocked in July. For those few sites, which we refer to as "sites to rerun", we can rerun AutoFR; this takes 1.6 min-per-site on average.

Site Snapshots Over Time. We recollect site snapshots for our entire *W09-Dataset* in July 2022 and associate them with the results of re-applying the rules above. For the 6% of sites that AutoFR needs to rerun, we report the changes in their corresponding snapshots. Fig. 4.10 reports the changes in snapshots of the same site between January and July in terms of different nodes, edges, and URLs. It also compares the differences for all sites, with those 6% sites to rerun AutoFR. For all other sites, 50% and 70% of sites have more than ±1K changes in nodes and edges, respectively; while 40% of sites have more than ±100 changes in URL nodes. Compared to sites to rerun, 75% of sites have more than ±1K changes in nodes and edges, while 65% of sites have more than ±100 changes in URL nodes. As expected, the

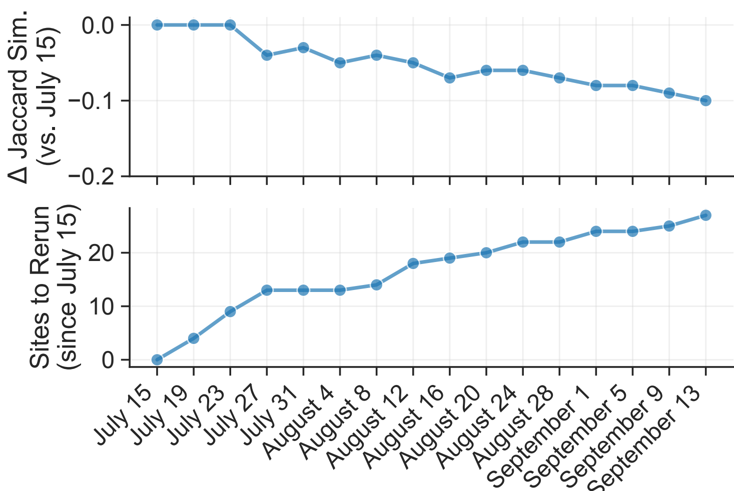


Figure 4.11: **Longitudinal Study Every Four Days.** We conduct a longitudinal study of 100 sites over a two-month period. We find that over time, site snapshots will become less similar (*i.e.,* negative Δ Jaccard similarity), denoting that rules are less effective. FL authors can rerun AutoFR on these sites that change more frequently to output effective rules.

snapshots of the sites to rerun indeed change more than other sites. However, AutoFR's rules remain effective on the vast majority of sites whose snapshots do not significantly change.

Why do Rules become Ineffective? For the sites that need to be rerun, we conduct a comparative analysis of how rules change by rerunning AutoFR on those sites. We find that 23% of these sites have completely new rules than before, which is typically due to a change in ad-serving infrastructure on the site. 40% of the sites need some additional rules (some older rules still work), which is due to additional ad slots on the site. In addition, 9% of the sites have changes in their paths. Lastly, 29% of these sites have the same rules as before. We deduce that this is because the rules are the best we can do without pushing breakage beyond the acceptable threshold w.

Takeaways. AutoFR rules need to be updated for a small fraction of sites (6% of Top–5K in six months), which demonstrates that AutoFR generates robust rules over time. AutoFR can be rerun for these sites at an average of 1.6 min-per-site.

4.5.3.2 How Frequently Should We Run AutoFR?

Next, to understand how often FL authors should run AutoFR over time, we provide a finer-grain longitudinal study of every four days for two months to study how site snapshots change and the sites that need AutoFR to be rerun. We choose every four days because this is how often EasyList is updated and deployed to end-users. In addition, we choose to focus on 100 sites, two-thirds of which are sampled from *W09-Dataset* and one-third is sampled from the set of 6% of sites that need to rerun in July (from Sec. 4.5.3.1). Fig. 4.11 illustrates our two-month results, using July 15, 2022, as our baseline. In this study, using Jaccard similarity, our comparison considers the relationship between HTML, JS, and CSS (different nodes within site snapshots). To do so, we retrieve the path from the root to every URL node for every site snapshot. We then convert these paths to strings and use them to calculate the Jaccard similarity between the site snapshots of July 15 to subsequent dates shown in the figure.

As expected, we arrive at the same conclusion as Sec. 4.5.3.1. Over time, the similarity between site snapshots will naturally decrease, denoting that there are sites where our rules are no longer effective, and we need to rerun AutoFR on them. For our 100 sites, we ran AutoFR on 13 sites once (*e.g., weheartit.com*, *legit.ng*), three sites twice (*e.g., buzzfeednews.com*), and two sites three or more times (*e.g., npr.org*), within two months. In terms of the time between the reruns of AutoFR, we find that one site (*e.g., charlotteobserver.com*) varied between four to 10 days from August 12 to September 13. This was due to path changes that would evade our rules[5] Similarly, one site (*e.g., npr.org*) varied from two weeks to one month. In addition, two sites had runs that were 1–2 weeks apart (*e.g.,* AutoFR found additional rules for *amarujala.com*). Lastly, one site had runs that were a month apart (*e.g., liputan6.com* went from *||googlesyndication.com^* to a new rule, *||infeed.id^*). By the end of this study, the similarity of site snapshots decreased by 10% (compared to site snapshots of July 15), and we ran AutoFR 27 times on 18 unique sites within two months.

[5] *e.g., ||charlotteobserver.com/.../0a086549941921c9ac8e.js*

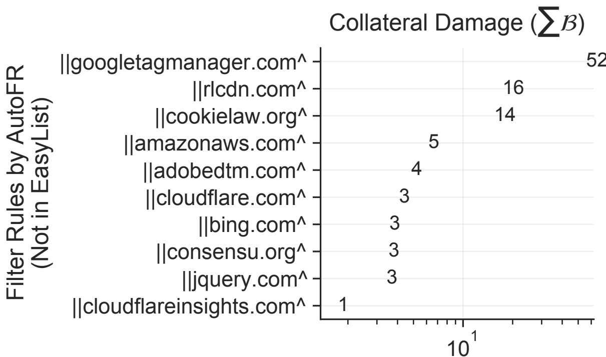


Figure 4.12: **Collateral Damage of Global Rules.** AutoFR rules are generated per-site and can potentially cause breakage when applied to other sites (*i.e.,* treated as a global rule). We report the rules that are unique to AutoFR (*i.e.,* not part of EasyList), ordered by decreasing total collateral damage ($\sum\mathcal{B}$) that they cause to site snapshots within *Full-W09-Dataset*. We can see that most of these rules (93%) cause negligible collateral damage (below 10 on the x-axis). Note that the possible max $\sum\mathcal{B}$ of each rule is the size of the dataset.

Takeaways. We find that each site will naturally change over time, causing site snapshots to be less similar. More changes often denote a higher possibility of rules being evaded. Overall, 18% of 100 sites needed a rerun of AutoFR. FL authors can periodically rerun AutoFR on sites that tend to change frequently in terms of weekly to monthly reruns. AutoFR minimizes the human effort for updating rules over time.

4.5.3.3 Evading URL-based Filter Rules

AutoFR generates URL-based filter rules, which EasyList also supports. Well-known evasion techniques for URL-based filter rules, such as randomizing URL components, affect both AutoFR rules and EasyList rules [85]. The strength of AutoFR is that new rules can be learned automatically and quickly (*e.g.,* in 1.6 min-per-site on average) when old ones are evaded. Publishers and advertisers can also try to specifically evade AutoFR [85, 130]. For example, they can put ads outside of iframes, use different ad transparency logos, or split the logo into smaller images, preventing Ad Highlighter from detecting ads [130]. This impacts our reward calculations. Defense approaches include the following. At the component level, we can try to improve Ad Highlighter to handle new logos or look beyond iframes, replace

		Methodology	**Input** (n sites)	**Output** (rules)	**Does Not Need Existing Rules**	**Generalizes**	**Efficiency** (min) (empirical) (Sec. 4.6.3.2)	**Maintainability** (update rules for m known sites) (Sec. 4.6.3.2)
Construction	1	AutoFR (Sec. 4.5)	1	Per-site	✓	×	1.6	Rerun AutoFR for m sites
Construction	2	AutoFR-Global (Sec 4.6.2.1)	1+	Global	✓	✓	$[0.8 \times n] + n$	Collect snapshots for m sites and rerun AutoFR-Global.
Post-process	3	AutoFR-Pop (Sec. 4.6.2.2)	1+	Global	×	✓	$[1.6 \times n]$	Rerun AutoFR for m sites and run post-processing
Post-process	4	AutoFR-Sim (Sec 4.6.2.3)	1+	Per-site	×	✓	$[1.6 \times n] + 0.8$	Rerun AutoFR for m sites and run post-processing.

Table 4.5: **Generating Filter Rules Across Multiple Sites.** We compare different approaches for generating filter rules. We split them into two categories. "Construction" approaches optimize rules during the training process, while the others apply a "post-processing" step on existing per-site rules. We use AutoFR (row 1) as our baseline. The column "Generalizes" denotes whether the approach can deal with unseen sites. Efficiency provides empirical estimates of each approach in minutes. Square brackets [ ] denote that parallalization can be used to remove n, *e.g.,* $1.6 \times n \rightarrow 1.6$.

Ad Highlighter with a better future visual perception tool, or pre-process the logos to remove adversarial perturbations [75]. At the system level, as an adversarial bandits problem, where the reward received from pulling an arm comes from an adversary [23].

4.6 Generating Rules Across Multiple Sites

By design, the AutoFR framework optimizes per-site rules. We now address the problem of generating rules across multiple sites by developing new approaches that consider both per-site and global rules. These approaches either leverage AutoFR as a building block or extend the AutoFR framework. Using AutoFR as a baseline, we compare these approaches and evaluate their effectiveness at blocking ads *vs.* avoiding breakage to known sites, and how well they generalize to unseen sites. Sec. 4.6.1 compares the differences between per-site

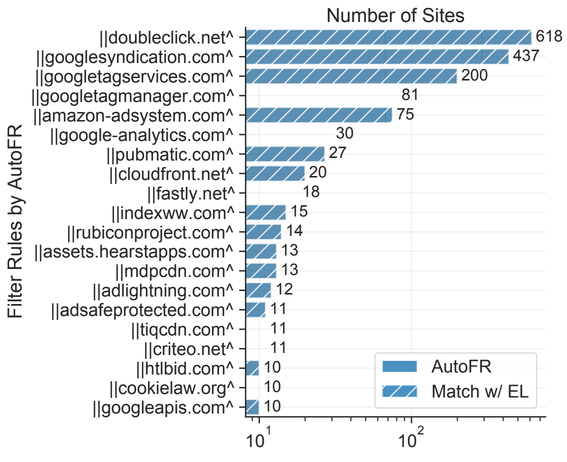
and global rules. Sec. 4.6.2 outlines our approaches to generate filter rules across multiple sites. Sec. 4.6.3 presents their comparative analysis and makes recommendations.

4.6.1 Per-site vs. Global Filter Rules

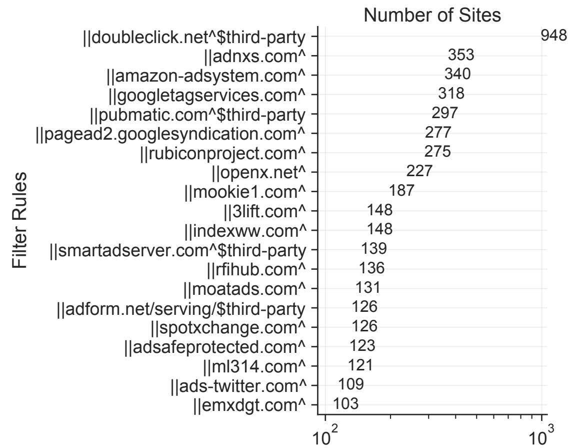
Recall that Sec. 2.1.2 and Table 2.1 introduce ways in which filter rules can be applied to known and unseen sites, and their potential for collateral damage. The first are per-site rules, which can only trigger for known sites. For example, AutoFR generates per-site rules for *cnn.com*, which can only be used on *cnn.com*, *e.g.,* ||*ad.com*ˆ$*domain*=*cnn.com*. Note that the previous sections disregard this qualifier for simplicity when mentioning per-site rules. The second, which we refer to as "global" filter rules, can trigger for both known and unseen sites, *e.g.,* ||*ad.com*ˆ. Thus, these definitions are based on how they can be applied to sites and not the methodology used to generate them.

EasyList supports per-site rules and currently contains ~800 of them. Per-site rules are guaranteed to perform well on known sites because they are commonly optimized for those sites (Sec. 4.5.1). However, as they cannot be applied to unseen sites, they cause the filter list to be larger. More rules in a filter list mean more effort to maintain them over time. On the other hand, global filter rules do not have these limitations. Hence, EasyList contains mostly global rules. However, when rules are applied to unseen sites, they have the potential to cause collateral damage.

In Fig. 4.12, we report the potential collateral damage, defined as the sum of breakage ($\sum\mathcal{B}$), caused when AutoFR per-site rules are treated as global rules. This serves as an example of applying per-site and global rules to unseen sites. We observe that they tend to block tag managers (*e.g.,* ||*googletagmanager.com*ˆ, ||*adobedtm.com*ˆ), CDNs or cloud storage services (*e.g.,* ||*cloudflare.com*ˆ, ||*amazonaws.com*ˆ, ||*rlcdn.com*ˆ), third-party libraries (*e.g.,* ||*jquery.com*ˆ), and cookie consent forms (*e.g.,* ||*cookiekaw.org*ˆ, ||*consensu.org*ˆ). These



(a) **Top–20 Filter Rules by AutoFR for Top–5K Sites.** They include the main advertising and tracking services, such as Alphabet and Amazon. They are likely to generalize well.



(b) **Top–20 Filter Rules by EasyList for Top–5K Sites.** We apply EasyList to the same Top–5K sites in Sec. 4.5.1 and show the popular rules by the number of sites that they "hit" on.

Figure 4.13: **AutoFR** *vs.* **EasyList: Popular Rules**

rules target domains that can serve legitimate content and ads across different sites. Thus, adopting a per-site rule into a global rule is nontrivial because the rule may not block as many ads or may cause more breakage (*i.e.,* collateral damage). It is not a problem distinct to AutoFR. Our discussions with EasyList authors confirmed that new rules are created per-site. They become global rules when FL authors know that the same rules are effective for other sites. FL authors rely on feedback from users to know when global rules either are ineffective or cause collateral damage on unknown sites [19].

Global rules can be evaluated by: (1) how a rule affects multiple sites using the sum of $\widehat{C}_A$, $\widehat{C}_I$, $\widehat{C}_T$ (Eq. (4.1)) to calculate breakage B (Eq. (4.2)), shown in Fig. 4.12; (2) how a set of rules affect multiple sites using averages of $\widehat{C}_A$, $\widehat{C}_I$, $\widehat{C}_T$ (Eq. (4.4)) and $\mathcal{R}_F$ (Eq. (4.5)), shown in Fig. 4.14; and (3) the number of sites they are effective on, shown in Tables 4.6, 4.7.

4.6.2 Methodologies to Generating Filter Rules

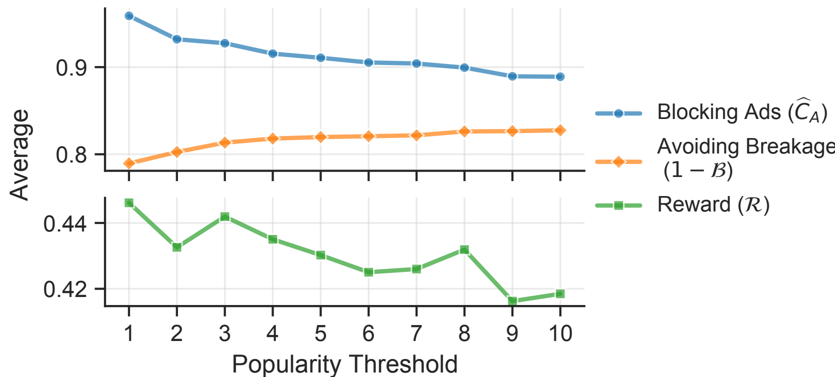


Figure 4.14: **Selecting Per-Site Rules into Global Filter Lists.** After creating the per-site AutoFR rules for each site (with $w = 0.9$), we create 10 global filter lists. "Popularity 1" means that a rule is selected into the global list if it was generated in at least one site; "Popularity 10" means that a rule is selected if it was generated for at least 10 sites. Once selected, the rules are now treated as global rules. We apply these global filter lists on our *Full-W09-Dataset* site snapshots and plot the average blocking ads, avoiding breakage, and reward.

There are two general approaches to generating filter rules for both known and unseen sites. The first approach is based on post-processing of per-site rules, *i.e.,* it selects and possibly modifies from existing per-site rules to also apply to other sites. The second possible approach is to optimize the rules during their construction (*i.e.,* training) process for a given set of sites and does not depend on existing per-site rules. In this section, we explore representative implementations of both of these broad approaches and summarize them in Table 4.5. We evaluate their performance, efficiency, and maintainability in Sec. 4.6.3.

4.6.2.1 AutoFR-Global: Extending AutoFR for Global Filter Rules

For AutoFR-Global, we revise our Algorithm 1 to generate global rules (*i.e.,* output) optimized for a set of known sites (*i.e.,* input). By doing so, the output of possible global rules is not limited to the union of all known per-site rules, as compared to Sec. 4.6.2.2, 4.6.2.3. We make the following changes to the algorithm and refer to specific lines of code. First, we build our action space (line 4) from web requests of all visits to the given sites (*i.e.,* there is only one action space for the given sites). The virtual node of the action space simply represents all of the given sites instead of one site as before. Next, once an action is selected, we apply

the action to each site (lines 16–23). Similar to Algorithm 1, we consider rewards where the action caused at least one hit. Otherwise, we put the arm to sleep. Everything else, including hyper-parameters, is kept the same. The outputs are global rules optimized for the given sites. Note that if AutoFR-Global is given one site as input, it is essentially AutoFR.

To optimize AutoFR-Global, we explore different ways of calculating the reward for the action. One approach is to take the site feedback terms ($\widehat{C}_A$, $\widehat{C}_I$, $\widehat{C}_T$)) (Eq. (4.1)) from applying the action to each site, then averages them before calculating the reward using Eq. (4.3).

$$\widehat{C}_A = \frac{1}{n}\sum_{i=1}^{n}\widehat{C}_{A_{\ell i}}, \quad \widehat{C}_I = \frac{1}{n}\sum_{i=1}^{n}\widehat{C}_{I_{\ell i}}, \quad \widehat{C}_T = \frac{1}{n}\sum_{i=1}^{n}\widehat{C}_{T_{\ell i}} \tag{4.4}$$

Another approach takes the individual rewards received after applying the action to each site and calculates their average. We define this average reward below, where ℓi represents a particular site and n is the number of sites:

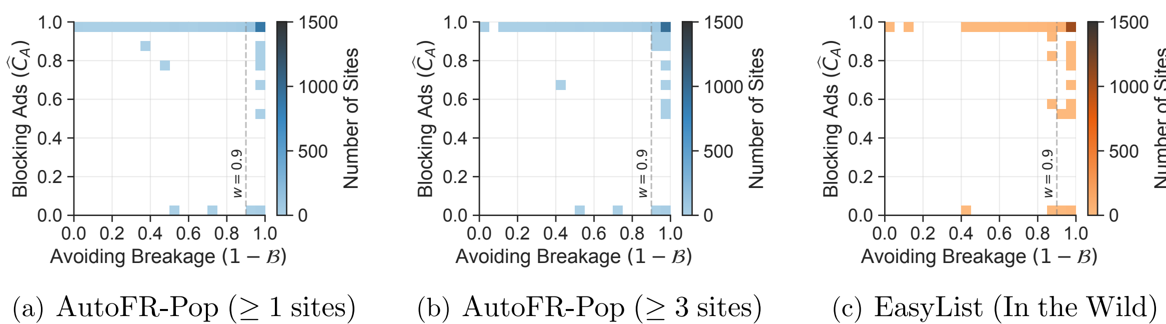
$$\mathcal{R}_F = \frac{1}{n}\sum_{i=1}^{n}R_{\ell i} \tag{4.5}$$

4.6.2.2 AutoFR-Pop: Using Popularity to Select Per-site Rules as Global Rules

Although we cannot guarantee, in advance, how well per-site rules will perform on other sites, we can try heuristics and assess their performance. Intuitively, if the same filter rule is generated by AutoFR across multiple sites, then it has a better chance of generalizing to new sites. As mentioned in Sec. 4.6.1, this is exactly the process that FL authors utilize when changing a per-site rule into a global rule. We denote this as the "popularity" of a rule. Fig. 4.13(a) shows the Top–20 AutoFR most popular rules across sites in the Top–5K, which shares common rules with EasyList in Fig. 4.13(b). They intuitively make sense as they belong to widely used advertising and tracking services. Therefore, we utilize this heuristic as criteria to select AutoFR per-site rules to include in filter lists. Once selected,

	Description ($w=0.9$)	AutoFR-Pop (≥ 1 sites) 1	AutoFR-Pop (≥ 3 sites) 2	EasyList 3
1	Sites in operating point: $\widehat{C}_A \geq 0.95$, $1-\mathcal{B} \geq 0.95$	67%	73%	80%
2	Sites within w: $\widehat{C}_A > 0$, $1-\mathcal{B} \geq 0.9$	76%	80%	87%
3	Ads blocked within w: $\sum_\ell(\overline{C}_A \times \widehat{C}_A) \,/\, \sum_\ell \overline{C}_A$; $1-B \geq 0.9$	72%	80%	86%

Table 4.6: **AutoFR-Pop** (Top 5K–10K, In the Wild). We evaluate AutoFR-Pop on unseen sites. See Fig. 4.15.



(a) AutoFR-Pop (≥ 1 sites) (b) AutoFR-Pop (≥ 3 sites) (c) EasyList (In the Wild)

Figure 4.15: **AutoFR-Pop: Top 5K–10K, In the Wild.** We create two filter lists, Fig. 4.15(a) with all rules from *W09-Dataset* and Fig. 4.15(b) that contains rules that were created for ≥ 3 sites. We test them in the wild on the Top–5K to 10K sites (unseen sites) and show their effectiveness along with EasyList (Fig. 4.15(c)). We observe that Fig. 4.15(b) performs better, blocking 8% more ads than Fig. 4.15(a). Table 4.6, col. 1–3, contains additional information.

we now treat them as global rules. To further understand how popularity can affect the performance of global filter rules, we leverage our site snapshots, as explained in Fig. 4.14. Notably, it depicts the trade-off between blocking ads *vs.* avoiding breakage, which exists even for global filter rules. As the popularity increases, the global filter list contains fewer global rules, resulting in fewer blocked ads but less breakage.

4.6.2.3 AutoFR-Sim: Applying Per-site Filter Rules from Similar Sites

We devise a collaborative filtering approach whereupon visiting an unseen site, we utilize per-site filter rules from similar sites. This is an intuitive approach: sites commonly share

similar eSLDs that are related to advertising and tracking services. Thus, if two sites contact roughly the same set of eSLDs, then the same filter rules will be effective for both. Other notions of similarity, can also be incorporated by our framework, if so desired. This is illustrated in Fig. 4.13. We envision the following scenario. First, there is a large dataset of known sites where we already applied AutoFR, such as our *W09-Dataset*. Second, when a user visits an unseen site, we will identify its Top–K similar sites within our dataset using a similarity metric. Then, we treat per-site rules of similar sites as per-site rules belonging to the unseen site. This strategy allows us to rely solely on per-site rules. Note that for known sites, we simply apply their corresponding per-site rules.

Motivated by how we measure site dynamics using unique eSLDs of a site in Sec. 4.5.1.1, we now represent a site as the set of unique destinations that it contacts to calculate the similarity between sites. For example, for any eSLD u and site ℓ_i in a training set $\mathcal{D}$, $\forall i \in \mathcal{D}$, we define $S_{\ell i}$ as the set of unique u, found in the site's action space. Then, their set union is $S_{\mathcal{D}} = \bigcup S_{\ell i}$. Then for every $u \in S_{\mathcal{D}}$, we can now represent a site ℓ as vector $\vec{X}_\ell = [x_{u1}, x_{u2}, ..., x_{u|S_{\mathcal{D}}|}]$, where $x_u = 1$ if $u \in S_\ell$ and zero otherwise. We can now leverage $\vec{X}_\ell$ to calculate similarity metrics as the cosine similarity of two sites:

$$\cos(\vec{X}_{\ell 1}, \vec{X}_{\ell 2}) = \frac{\vec{X}_{\ell 1} \cdot \vec{X}_{\ell 2}}{\left\|\vec{X}_{\ell 1}\right\| \left\|\vec{X}_{\ell 2}\right\|} \tag{4.6}$$

For the Top–K similar sites, we calculate the cosine distance: $d(\vec{X}_{\ell 1}, \vec{X}_{\ell 2}) = 1 - \cos(\vec{X}_{\ell 1}, \vec{X}_{\ell 2})$.

4.6.3 Evaluation

4.6.3.1 Evaluating AutoFR-Pop In the Wild

To understand how we can evaluate and compare different methodologies for generating filter rules across multiple sites, we first take AutoFR-Pop as a naive use case.

	Methodology	Training Set (466 sites)			Test Set (350 sites)			Recommendations				
		Operating point	Sites in w	Ads blocked in w	Operating point	Sites in w	Ads blocked in w	Performance (Training)	Generalization (Test)	Efficiency	Maintainability	Best Overall
		1	2	3	4	5	6	7	8	9	10	11
1	AutoFR (Baseline)	59%	77%	76%	64%	78%	75%	●	○	●	●	◉
AutoFR-Global												
2(a)	Eq. (4.4) and Eq. (4.3)	40%	53%	56%	42%	54%	53%	◐	⊙	⊙	⊙	⊙
2(b)	Eq. (4.5)	51%	69%	67%	50%	64%	58%	◉	◐	⊙	⊙	◐
AutoFR-Pop												
3(a)	$\geq$1 sites	41%	53%	54%	42%	53%	52%	⊙	⊙	◉	◉	◐
3(b)	$\geq$2 sites	41%	53%	54%	44%	54%	53%	⊙	⊙	◉	◉	◐
3(c)	$\geq$3 sites	42%	55%	57%	44%	54%	53%	◐	⊙	◉	◉	◐
3(d)	$\geq$4 sites	42%	56%	57%	45%	55%	53%	◐	⊙	◉	◉	◐
3(e)	$\geq$5 sites	42%	57%	58%	46%	56%	54%	◐	⊙	◉	◉	◐
3(f)	$\geq$6 sites	45%	61%	60%	49%	60%	59%	◐	◐	◉	◉	◐
3(g)	$\geq$7 sites	44%	61%	59%	49%	60%	59%	◐	◐	◉	◉	◐
3(h)	$\geq$8 sites	44%	61%	59%	49%	60%	59%	◐	◐	◉	◉	◐
3(i)	$\geq$9 sutes	44%	61%	59%	49%	60%	59%	◐	◐	◉	◉	◐
3(j)	$\geq$10 sites	44%	61%	59%	49%	60%	59%	◐	◐	◉	◉	◐
AutoFR-Sim (eSLD, cosine distance d)												
4(a)	Top–1, $d \leq 0.2$	59%	77%	76%	19%	23%	22%	●	○	◐	◉	◐
4(b)	Top–3, $d \leq 0.2$	59%	77%	76%	19%	23%	24%	●	○	◐	◉	◐
4(c)	Top–5, $d \leq 0.2$	59%	77%	76%	19%	23%	24%	●	○	◐	◉	◐
4(d)	Top–1, $d \leq 0.6$	59%	77%	76%	48%	62%	60%	●	◐	◐	◉	◐
4(e)	Top–3, $d \leq 0.6$	59%	77%	76%	53%	66%	65%	●	◐	◐	◉	◐
4(f)	Top–5, $d \leq 0.6$	59%	77%	76%	53%	65%	64%	●	◐	◐	◉	◐
4(g)	Top–1, $d \leq 1.0$	59%	77%	76%	48%	62%	60%	●	◐	◐	◉	◐
4(h)	Top–3, $d \leq 1.0$	59%	77%	76%	53%	66%	65%	●	◐	◐	◉	◐
4(i)	Top–5, $d \leq 1.0$	59%	77%	76%	53%	65%	64%	●	◐	◐	◉	◐

Table 4.7: **Generating Rules across Multiple Sites (using Site Snapshots).** We leverage our *W09-Dataset* to evaluate our methodologies (from Table 4.5) for generating filter rules that can be applied across multiple sites. We provide recommendations by comparing the approaches based on performance on the known sites (training set), how well they generalize to unseen sites (test set), and their empirical efficiency and maintainability (over time). We use the following criteria for each methodology from col. 7–8: ○ = 30+% from the baseline, ⊙ = 30%, ◐ = 20%, ◉ = 10%, ● = same as baseline. On the other hand, "Best Overall" treats ● = 1. We then sum up the scores of each row from col. 7–10 and take their averages. *e.g.,* row 1 has ◉ = 3/4.

We analyze in detail two global filter lists. First, "popularity 1" (*i.e.,* $\geq$ 1 sites) treats all AutoFR per-site rules as global rules, which serves as a baseline for comparison. Second, "popularity 3" denotes AutoFR rules that were generated from $\geq$ 3 sites. Fig. 4.14 reveals

that this has the highest average reward. Note that selecting the popularity threshold based on the average reward implicitly considers collateral damage because it encompasses breakage (Eq. (4.3)). We apply these global filter lists on the Tranco Top 5K–10K sites in the wild. Fig. 4.15 and Table 4.6 col. 1–3 show the results. As expected, we see that the global filter list created from rules that appeared in ≥ 3 sites perform better than the list with all rules. Moreover, Fig. 4.15(b) compares relatively well against Fig. 4.15(c) (EasyList): 73% of sites are in the desired operating point (top-right corner), *vs.* 80% by EasyList (row 1, col. 2–3). Overall, the rules generated from the Top–5K sites were able to block 80% of ads on the Top 5K–10K sites. This shows a good generalization of AutoFR rules across unseen sites, which agrees with Fig. 4.13(a). However, although expected, there is a decrease in performance when applied to unseen sites. Motivated by this use case, we will evaluate all methodologies on known sites and unseen sites in the next section.

4.6.3.2 Known Sites vs. Unseen Sites

To evaluate each methodology described in Sec. 4.6.2, we split our *W09-Dataset* into a training set by randomly sampling 50% of the dataset (466 sites). We treat the remaining as a test set (350), *i.e.,* the unseen sites. We remove any shared eSLDs from the test set that it has with the training set so that similarity between home and sub-pages is not a factor. Table 4.7 presents our comparative results across performance, efficiency, and maintainability factors, as discussed in Sec. 2.1.2. This corresponds to Table 4.5.

1. *AutoFR* (row 1): For the training set, we apply the per-site rules to their specific site. We do the same for the test set (col. 4–6). This serves as our baseline to compare against other approaches.

2. *AutoFR-Pop* (row 2(a)–2(b)): For the training set, we take the per-site rules created by AutoFR (row 1) and select them as global rules within a filter list using a popularity threshold of 1–10. We then apply those same filter lists generated by the training set

to the test set.

3. *AutoFR-Global* (row 3(a)–3(j)): We explore two ways of calculating rewards as described in Sec. 4.6.2.1. For the test set, we apply the generated global rules created during the training phase to the test set.

4. *AutoFR-Sim* (row 4(a)–4(r)): We evaluate AutoFR-Sim using three different factors: (a) representing a site as a vector of eSLDs or FQDNs; (b) which Top–K to use for a range of 1–5; and (c) a minimum distance threshold to be considered a Top–K. For instance, for row 4(a), the distance between site ℓ_1 and ℓ_2 must be ≤ 0.2 before using the filter rules. For brevity, we report the results for cosine distance only and Top–[1, 3, 5].

Performance (Training Set). We now examine the results in Table 4.7 using AutoFR as our baseline. We first focus on the training set results. AutoFR-Pop performs identically to our baseline by returning the known per-site rules. A close second is AutoFR-Global using Eq. (4.5), which outperforms AutoFR-Pop. This is not surprising, as AutoFR-Global is not confined to existing per-site rules like AutoFR-Pop. For AutoFR-Pop, the popularity threshold ≥ 6 performs the best among the different thresholds. Although, we note that it did not perform considerably better than other thresholds from ≥ 7 to ≥ 10.

Generalization (Test Set). Next, we focus on the test set results. We find that AutoFR-Sim performs the best, especially for Top–[3, 5] with d ≤ 0.6. AutoFR-Global and AutoFR-Pop come in a close second. Notably, all of the approaches do not closely compare to our baseline. This highlights the advantage of optimizing filter rules per-site and further illustrates the limitation of generalizing rules to unseen sites. As expected, we cannot guarantee good performance for our test set.

Efficiency. Table 4.5 provides the efficiency of each approach using empirical estimates using our Python-based implementation, while Table 4.7 visualizes it. AutoFR-Pop runs AutoFR for every given site to have per-site rules before selecting them as global rules using

a predetermined threshold. We treat its post-processing as negligible time. For AutoFR-Global, it collects site snapshots of the given sites first, which takes on average 0.8 min per site (Sec. 4.4.1). Then, it runs the modified AutoFR algorithm. Similar to AutoFR, AutoFR-Global's efficiency is affected by the given sites and the reward function. The former affects the size of the action space, while the latter affects how much of the action space is explored. We execute AutoFR-Global for various input size (n sites) [1, 10, 50, 100, 200, 500] using Eq. (4.5) and find that it linearly increases with n. For AutoFR-Sim, we must run AutoFR for the given sites (like AutoFR-Pop) first. If an unseen site is given, then we must visit the site to collect the URLs that it contacts to build its vector representation. This takes on average 0.8 min (Sec. 4.4). Next, we must calculate the Top–K similar sites. Our experiments show that this takes on average 0.02–0.06 sec. We regard this as negligible time. AutoFR-Global cannot be optimized across actions. For example, if 100 sites were given to AutoFR-Global, at each time step t, it must finish applying the action to 100 sites first before selecting the next action at $t+1$. This can considerably slow down the run time.

Maintainability. Table 4.5 summarizes how to deal with maintaining filter rules over time, *i.e.,* when rules are no longer effective for some sites, while Table 4.7 visualizes the comparison. Most approaches rely on updating rules for the affected sites before applying some post-processing. However, rerunning AutoFR-Global is the most costly, especially if the number of affected sites is small. For instance, it may not be worthwhile to rerun AutoFR-Global when only one site has rules that are no longer effective. AutoFR is the easiest to maintain, followed by AutoFR-Pop and AutoFR-Sim.

Recommendations. As illustrated in Table 4.7, every possible approach has its own tradeoffs, including ones that we designed and implemented. For users who want to adopt our approaches to generating filter rules, we recommend the following. If we only care about the performance for a known set of sites without generalization, choose AutoFR. If we consider all factors (*e.g.,* performance, generalization, efficiency, and maintainability), choose

AutoFR-Sim or AutoFR-Pop. AutoFR-Sim's downside is that it delays the loading of the site during the post-processing step. However, this can be addressed with implementation, such as using open-sourced web archives to get the URLs quickly for the site. To avoid this potential delay, choose AutoFR-Pop. Otherwise, if we care about performance and generalization only, choose AutoFR-Global or AutoFR-Sim.

Directions for Future Work. In this section, we compared the performance of four possible approaches to generating filter rules using site snapshots. Future work can explore their efficacy in the wild. Furthermore, although we compare them as separate methodologies, hybrid approaches may yield better results. For instance, one can use AutoFR-Sim for unseen sites while triggering the run of AutoFR for them behind the scenes. Once AutoFR is done, use the newly generated per-site rules instead. Conversely, if we want to utilize global rules instead, we can run AutoFR-Global infrequently (*e.g.,* once a month). To address its maintainability, run AutoFR for sites that have ineffective rules and use the per-site rules temporarily (for those sites) until the next run of AutoFR-Global.

4.7 AutoFR in a Live Environment (AutoFR-L)

In this section, we provide the full details of an alternative implementation of AutoFR in a live setting, referred to as AutoFR-L. Sec. 4.7.1 explains the differences between AutoFR (that runs using site snapshots) *vs.* AutoFR-L (that visits the site for real during the initialize phase or when an arm is pulled). In Sec. 4.7.2, we detail the implementation. Fig. 4.16 illustrates an example of how AutoFR-L works end-to-end.

4.7.1 AutoFR vs. AutoFR-L

This section complements Sec. 4.4, where we presented the implementation of AutoFR in a controlled environment (*i.e.,* based on site snapshots). There, we argued that an imple-

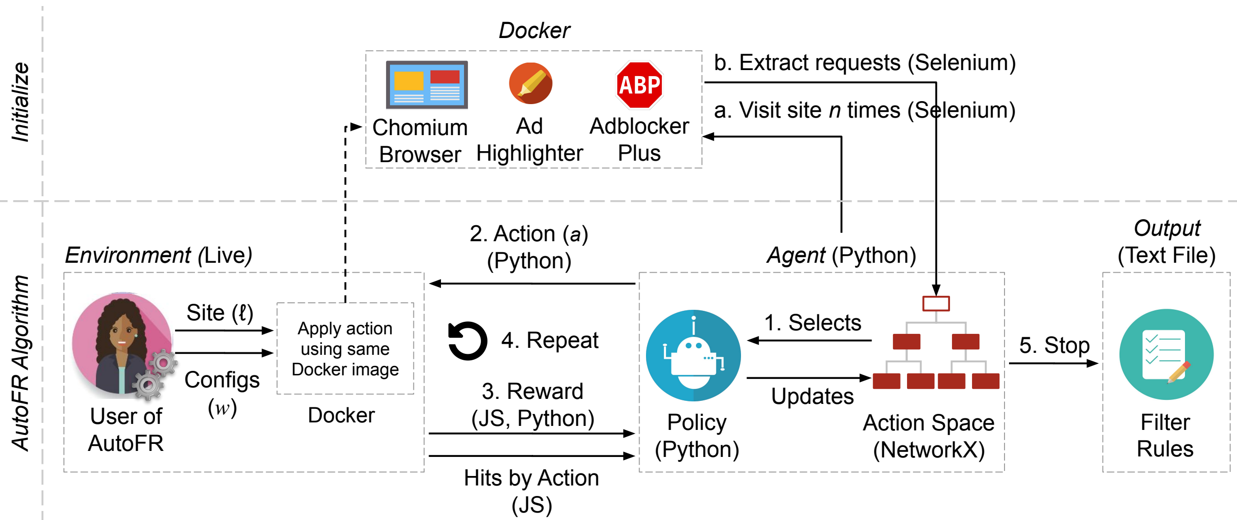


Figure 4.16: **AutoFR-L Example Workflow (Live Environment).** INITIALIZE (a–b, Alg. 1): (a) spawns $n = 10$ docker instances and visits the site until it finishes loading; (b) extracts the outgoing requests from all visits and builds the action space. We run the RL portion of AUTOFR procedure (steps 1–4). Lastly, AutoFR outputs the filter rules at step 5, *e.g.,* ||*s.yimg.com/rq/darla/4-10-0/html/r-sf.html*. Note that we do not use AdGraph or site snapshots in this version.

mentation of AutoFR that exactly mimics the human process, would need to interact with sites and test different rules in a live environment, which would be slow and expensive, albeit strictly better than the human FL author process. In this appendix, we describe the implementation of this live version, which we refer to as AutoFR-L. It is worth emphasizing that the distinction between controlled (*i.e.,* based on snapshots) and "live" in the implementation of AutoFR applies only to the *training* phase, *i.e.,* during the trial and evaluation of candidate filter rules. Once the filter rules are generated with either version of the implementation, they can be *applied or tested* on any site in the wild. Fig. 4.16 outlines how we implement AutoFR in a live environment (AutoFR-L). This means that AutoFR-L visits the site for real at every time step t of the algorithm. It corresponds to our formulation of the problem in Fig. 4.1(b). To simplify our explanation, we follow the same outline as in Sec. 4.4.

4.7.2 AutoFR-L Implementation

Agent. The implementation of the agent, policy, and action space is the same as Sec. 4.4.2.

Environment (Live). The environment allows the agent to apply an action in a live setting. In particular, it visits a site ℓ for real and applies the filter rule using Adblock Plus [10]. It then captures the site feedback (*e.g.,* ads, images, text) using JS injection and calculates the reward, and returns it back to the agent. A visit to a site for real is explained in Sec. 4.7.2.2. Importantly, it has a high cost and we deem it impractical, as discussed in Sec. 4.4.

4.7.2.1 Ad Highlighter

As discussed in Sec. 4.4.3, we rely on Ad Highlighter [127] to capture the number of ads during a visit to a site. Ad Highlighter works in the following ways. Within every iframe, it finds all images and SVGs, or HTML elements that contain the background-url style (*e.g.,* spans, a tags, and divs), and calculates their 625-bit image hash. It calculates a Jaccard similarity score between the image hash and the set of known hashes (hard-coded), if the similarity is above 0.8, it marks it as an ad by overlaying it with the word "AdChoice Identified." We modify Ad Highlighter to listen to a custom event so that we can extract the number of ads it identified using JS injection. Ad Highlighter is easily extendable using JS. It also allows us to audit its effectiveness visually using a browser. In addition, the similarity matching threshold is easily tuned to improve the precision of the tool. Ad Highlighter has high precision; we explain this in Sec. 4.5.1.4.

4.7.2.2 Visiting a (Live) Site

This involves the same setup as described in Sec. 4.4.1 when collecting raw AdGraphs. However, we use the Chromium browser this time with Ad Highlighter and a customized ABP (taken from [85]). We use Selenium to toggle off any filter lists that are by default loaded.

As a result, ABP will start off with no filter rules loaded.

Applying Action a. To apply our action (filter rule), we do the following methodology as [85]. First, we use Flask, a Python web server to locally serve our custom filter rules. We then utilize a customized ABP extension with the browser [85]. This allows us to load the custom filter rules being served by Flask. In addition, we can retrieve the filter rules that block any outgoing requests (*i.e.,* hits). To do so, we trigger a custom event that ABP responds to. ABP will add the *hits* information in JSON format into the body of the page. We then retrieve it from the page by injecting JS using Selenium.

Capturing Site Feedback. To capture the number of ads (C_A), we rely on Ad Highlighter, as described in Sec. 4.7.2.1. For C_I, we inject custom JS to retrieve images, similar to [85]. It considers all visible images (*e.g.,* with height and width $>$ 2px and opacity $>$ 0.1) or HTML elements that have a background-url set [99]. Similarly, for C_T, we inject custom JS to find all visible textnodes, which are locations of texts and not the individual word count [91]. This allows us to deal with dynamic sites that can serve personalized content. For example, a news site can display the same layout for five articles, each article contains one title and one description. The articles may change upon separate visits, but our approach allows us to still retrieve the same number for text (*e.g.,* $C_T = 5 \times 2$) without worrying about the change in the content itself. Using Selenium, we are able to collect the outgoing network requests ($reqs$) from the browser, including the initiator information, a call stack that connects how scripts call each other, and also which script initiated the request.

Rewards. We calculate rewards with python, as explained in Sec. 4.3.2.2.

4.8 Conclusion & Future Directions

Summary. The filter list curation follows a human-in-the-loop approach: (1) the rules are manually created, visually evaluated, and maintained; and (2) the FL author has to carefully balance between blocking ads *vs.* avoiding breakage. We introduced the AutoFR framework to automate the process of generating URL-based filter rules to block ads from scratch. Our implementation of the framework allows it to learn rules without relying on existing rules created by humans. Our evaluation showed that AutoFR is efficient and performs comparably to EasyList. Next, we develop and evaluate new approaches to generating filter rules that apply across multiple sites. We envision that AutoFR will be used by the adblocking community to automatically generate and update filter rules at scale.

Future Directions. AutoFR provides a general framework for automating filter rule generation. In this paper, we focused specifically on the commonly used URL-based rules for blocking ads on browsers, but we envision several extensions and applications. The AutoFR framework can be extended to include: (1) the creation of global rules, in addition to site-specific rules, (2) rules that block tracking; (3) other types of filter rules, such as element hiding rules, *e.g.,* using the concept of CSS specificity to leverage the hierarchy; (4) functionality (beyond visual) breakage, *e.g.,* by testing click functionality for buttons and links; (5) new visual detection modules for images and ads on sites as these become available. The AutoFR implementation, generated filter rules, and the dataset are available at [83].

Chapter 5

Conclusion

5.1 Summary

In this dissertation, we address major challenges in the automation of creating and maintaining filter rules for web adblockers. In doing so, we improve the robustness and scalability of filter rules for adblocking users.

First, we conduct a longitudinal analysis to measure the accelerated arms race between publishers, advertisers and adblockers. We find that it takes substantial human effort to maintain filter rules. Furthermore, we investigate the state-of-the-art techniques that publishers employ to evade adblockers. Leveraging our understanding of these techniques, we develop CV-Inspector, a machine-learning tool based on differential analysis to detect when websites circumvent adblockers. Using CV-Inspector, filter list authors no longer need to manually check thousands of websites to know when filter rules need to be updated.

Second, we examine the fundamental problem of creating filter rules from scratch. We formulate this problem within a reinforcement learning framework, called AutoFR, using multi-arm bandits. Given a website, AutoFR optimizes and outputs filter rules that block ads for the

website while remaining within an acceptable breakage threshold the user selects. To achieve practicality and scalability, we implement AutoFR to run within a controlled but realistic environment using site snapshots that represent how websites are loaded as graphs. Next, using AutoFR as a building block, we develop new approaches that generate rules across multiple websites. AutoFR further minimizes the human involvement in creating filter rules, which makes it possible to scale filter rule generation to thousands of websites and over time. We envision that our findings, methodologies, and tools will be useful to the adblocking community.

Future Directions. AutoFR can be extended to other platforms, such as mobile, smart TVs, and VR devices, as there is a need for better platform-specific filter lists, in terms of coverage and breakage [121, 137, 132]. On mobile and smart TVs specifically, one could leverage existing tools to automatically explore apps or mobile browsers [121, 137, 88, 41]. Furthermore, it can be extended to generate rules that block tracking. Deploying AutoFR and its extensions creates a scalable and maintainable future for filter rules across platforms. These rules and their updates can be open-sourced in repositories for users (and researchers), comparable to current filter lists, such as EasyList. Lastly, there is a potential to apply and adapt the AutoFR framework for auditing streams of information. For instance, we can audit recommendation systems within social media platforms like TikTok for topics of interest, such as harmful and hateful content.

5.2 Perspective

Reflecting upon our work and the adblocking space, we anticipate the following future trends:

Accelerated Arms Race on Other Platforms: As automated approaches, such as AutoFR, are adopted for other platforms, their arms race will accelerate. Similar to what has happened on the web, we expect CV-Inspector to play a role in helping the adblocking community combat the circumvention of adblocking, especially through differential analysis

and the detection of randomizing URL components.

Filter Rules, AI, and Humans: We predict that filter rules will remain crucial for at least the next ten years because they are easy to interpret (the very rules tell the user what they are blocking) and they are easy to deploy (there are processes in place to push rule updates to users hourly, discussed in Sec. 3.3). New automated tools, like AutoFR, will further sustain the usefulness of filter rules for adblocking. However, we predict that advances in AI, such as GPT–4, can complement filter rules. For instance, GPT–4 can edit out ads within videos and then display them to the user [16]. In this case, using filter rules to block ads integrated into videos may not be possible, so a hybrid approach is necessary. Furthermore, GPT–4 can be utilized as a component within filter rule generation frameworks, like AutoFR. For instance, it can detect ads, site content, or even breakage. Lastly, human involvement in adblocking will be necessary as long as adblocking affects millions of users (*i.e.,* breakage must be minimized) and there is an arms race. However, new tools, such as CV-Inspector and AutoFR, will minimize human involvement while helping them focus on important cases.

Advances in Adblocking Cross-Device: We expect that adblocking across devices will improve. First, we can extend and employ AutoFR to generate optimized rules for specific platforms and apps, which is necessary to reduce their potential for breakage and improve their effectiveness [137]. Second, to apply these rules, adblocking tools, like Pi-Hole, can utilize network-based fingerprinting techniques to infer the device and the opened application. This overall approach will enhance privacy protection for users and make adblockers more prevalent beyond just the web browser.

Non-profit Initiatives for Adblocking: Adblocking companies are businesses. They earn revenue through donations and whitelisting programs (Sec. 2.2.1). Recently, some have even established their own ad exchanges like the Acceptable Ads Exchange [1]. In addition, privacy-focused browsers that integrate adblocking functionality have launched their own

first-party ad platform [36]. As users see more ads, we expect this will spawn new non-profit initiatives to block ads and tracking so that the service is disentangled from financial incentives: adblockers paid by ads – the very problem that adblockers are supposed to stop.

Bibliography

[1] AAX. Acceptable ads exchange. https://www.aax.media/, 2023. (Accessed on 07/23/2023).

[2] Z. Abi Din, P. Tigas, S. T. King, and B. Livshits. PERCIVAL: Making in-browser perceptual ad blocking practical with deep learning. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 387–400, Virtual, July 2020. USENIX Association.

[3] Acceptable Ads. Sustainable and nonintrusive advertising. https://acceptableads.com/. (Accessed on 01/27/2022).

[4] Acceptable Ads. The Acceptable Ads Standard. https://acceptableads.com/standard.

[5] Adblock Plus. Adblock warning removal list. https://easylist-downloads.adblockplus.org/antiadblockfilters.txt. (Accessed on 07/10/2020).

[6] Adblock Plus. Taboola whitelisting too annoying, turned off acceptable ads. https://adblockplus.org/forum/viewtopic.php?f=17&t=50287, January 2017. (Accessed on 05/04/2020).

[7] Adblock Plus. Sentinel is online. https://blog.adblockplus.org/blog/sentinel-is-online, 2018. Archived at https://perma.cc/RNV9-5M5B. (Accessed on 01/24/2022).

[8] Adblock Plus. ABP anti-circumvention filter list . https://github.com/abp-filters/abp-filters-anti-cv, 2019. (Accessed on 05/09/2019).

[9] Adblock Plus. Contributors to abp-filters/abp-filters-anti-cv. https://github.com/abp-filters/abp-filters-anti-cv/graphs/contributors, May 2020. (Accessed on 05/21/2020).

[10] Adblock Plus. The world's # 1 free ad blocker. https://adblockplus.org/, 2023. (Accessed on 07/11/2023).

[11] AdDefend. Anti-adblock platform - addefend.com. https://www.addefend.com/en/platform/#why-anti-adblock-inventory, 2020. (Accessed on 03/18/2020).

[12] AdGuard. Network-wide software for any os: Windows, macos, linux. https://adguard.com/en/adguard-home/overview.html. (Accessed on 01/04/2022).

[13] AdGuard. World's most advanced adblocker! https://adguard.com/en/welcome.html. (Accessed on 01/03/2022).

[14] AdGuard. AdGuard Scriptlets and Resources . https://github.com/AdguardTeam/Scriptlets, 2019. (Accessed on 05/09/2019).

[15] AdGuard. Adguardextra: Adguard extra is designed to solve complicated cases when regular ad blocking rules aren't enough. https://github.com/AdguardTeam/AdGuardExtra, 2020. (Accessed on 03/18/2020).

[16] AdGuard. Can gpt-4 block ads better than humans? we put an ai-powered ad blocker to the test. https://adguard.com/en/blog/chatgpt-ad-blocking-extension.html, July 2023. (Accessed on 07/23/2023).

[17] Admiral. Admiral launches one-click subscriptions and donations for publishers to help grow alternative revenue post-gdpr. https://blog.getadmiral.com/admiral-launches-subscriptions-donations-transact-publishers, 2020. (Accessed on 03/18/2020).

[18] AdThrive. Ad management and optimization for the world's best content creators. https://www.adthrive.com/, 2020. (Accessed on 07/20/2020).

[19] M. Alzirah, S. Zhu, Z. Xing, and G. Wang. Errors, misunderstandings, and attacks: Analyzing the crowdsourcing process of ad-blocking systems. In *Proceedings of the Internet Measurement Conference*, Amsterdam, Netherlands, Oct. 2019. ACM.

[20] Amazon. Amazon ec2 instance types - amazon web services. https://aws.amazon.com/ec2/instance-types/, 2020. (Accessed on 05/09/2020).

[21] Amazon. Amazon machine images (ami) - amazon elastic compute cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html, 2020. (Accessed on 05/09/2020).

[22] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[23] P. Auer and C.-K. Chiang. An algorithm with nearly optimal pseudo-regret for both stochastic and adversarial bandits. In *Conference on Learning Theory*, pages 116–120, New York, NY, June 2016. PMLR.

[24] Backlinko. Ad blockers usage and demographic statistics in 2022. https://backlinko.com/ad-blockers-users, 2022. Archived at https://perma.cc/BG5J-B3FS. (Accessed on 01/27/2022).

[25] C. Barrett. Filterlists. https://filterlists.com/, 2022. Archived at https://perma.cc/KE8N-S6DE. (Accessed on 01/27/2022).

[26] M. A. Bashir, S. Arshad, E. Kirda, W. Robertson, and C. Wilson. How tracking companies circumvented ad blockers using websockets. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, pages 471–477, New York, NY, USA, 2018. ACM.

[27] M. A. Bashir, S. Arshad, and C. Wilson. "recommended for you": A first look at content recommendation networks. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, page 17–24, New York, NY, USA, 2016. Association for Computing Machinery.

[28] M. A. Bashir, U. Farooq, M. Shahid, M. F. Zaffar, and C. Wilson. Quantity vs. quality: Evaluating user interest profiles using ad preference managers. In *The Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019. The Internet Society.

[29] Better Ads. The Better Ads Standards. `https://www.betterads.org/standards`.

[30] S. Bhagavatula, C. Dunn, C. Kanich, M. Gupta, and B. Ziebart. Leveraging machine learning to improve unwanted resource filtering. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, pages 95–102, New York, NY, USA, Nov. 2014. ACM.

[31] BlockAdBlock. Stop losing ad revenue. `https://blockadblock.com/`. (Accessed on 05/04/2020).

[32] Blockthrough. 2020 adblock report. `https://blockthrough.com/2020/02/06/2020-adblock-report-3/`, February 2020. (Accessed on 03/18/2020).

[33] T. Boroushaki, I. Perper, M. Nachin, A. Rodriguez, and F. Adib. Rfusion: Robotic grasping via rf-visual sensing and learning. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 192–205, Coimbra, Portugal, Nov. 2021. ACM.

[34] Brave. Ad block engine used in the brave browser for abp filter syntax based lists like easylist. `https://github.com/brave/ad-block`, May 2020. (Accessed on 06/21/2020).

[35] Brave. Pagegraph: Wiki. `https://github.com/brave/brave-browser/wiki/PageGraph`, 2022. Archived at `https://perma.cc/78Q9-4KQX`. (Accessed on 01/28/2022).

[36] Brave. Brave ads. `https://brave.com/brave-ads/`, 2023. (Accessed on 08/01/2023).

[37] Brave. What is brave rewards? `https://brave.com/brave-rewards/`, 2023. (Accessed on 07/20/2023).

[38] S. Bubeck, T. Wang, and N. Viswanathan. Multiple identifications in multi-armed bandits. In *Proceedings of the 30th International Conference on International Conference on Machine Learning*, pages 258–265, Atlanta, GA, June 2013. PMLR.

[39] California. California consumer privacy act (ccpa). `https://oag.ca.gov/privacy/ccpa`, May 2023. (Accessed on 07/19/2023).

[40] W. Cao, J. Li, Y. Tao, and Z. Li. On top-k selection in multi-armed bandits and hidden bipartite graphs. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28, Montreal, Canada, Dec. 2015. Curran Associates, Inc.

[41] D. Cassel, S.-C. Lin, A. Buraggina, W. Wang, A. Zhang, L. Bauer, H.-C. Hsiao, L. Jia, and T. Libert. Omnicrawl: Comprehensive measurement of web tracking with real desktop and mobile browsers. In *Proceedings on Privacy Enhancing Technologies*, volume 1, pages 227–252, Sydney, Australia, July 2022. Sciendo.

[42] Q. Chen, P. Snyder, B. Livshits, and A. Kapravelos. Improving web content blocking with event-loop-turn granularity javascript signatures, 2020.

[43] Q. Chen, P. Snyder, B. Livshits, and A. Kapravelos. Detecting filter list evasion with event-loop-turn granularity javascript signatures. In *IEEE Symposium on Security and Privacy (SP)*, pages 1715–1729, San Francisco, CA, May 2021. IEEE.

[44] Y. Chen. Tough sell: Why publisher 'turn-off-your-ad-blocker' messages are so polite - digiday. `https://digiday.com/media/tough-sell-publisher-turn-off-ad-blocker-messages-polite/`, April 2016. (Accessed on 05/04/2020).

[45] Chromium. Chromedriver. `https://sites.google.com/a/chromium.org/chromedriver/`. (Accessed on 05/09/2020).

[46] Chromium. Under the hood: How chrome's ad filtering works. `https://blog.chromium.org/2018/02/how-chromes-ad-filtering-works.html`, Febrary 2018. (Accessed on 05/04/2020).

[47] T. Claburn. Revealed: The naughty tricks used by web ads to bypass blockers. `https://www.theregister.co.uk/2017/08/11/ad_blocker_bypass_code/`, 2017. (Accessed on 05/09/2019).

[48] R. Cointepas. Cname cloaking, the dangerous disguise of third-party trackers. `https://medium.com/nextdns/cname-cloaking-the-dangerous-disguise-of-third-party-trackers-195205dc522a`, November 2019. (Accessed on 05/04/2020).

[49] H. Dao, J. Mazel, and K. Fukuda. Characterizing cname cloaking-based tracking on the web. *IEEE/IFIP TMA'20*, pages 1–9, 2020.

[50] M. Degeling and J. Nierhoff. Tracking and tricking a profiler: Automated measuring and influencing of bluekai's interest profiling. In *Proceedings of the Workshop on Privacy in the Electronic Society*, WPES'18, page 1–13, New York, NY, USA, 2018. ACM.

[51] DisconnectMe. Tracking services. `https://raw.githubusercontent.com/disconnectme/disconnect-tracking-protection/master/services.json`, May 2020. (Accessed on 06/21/2020).

[52] S. Dixit. Block, unblock, block! How ad blockers are being circumvented . `https://www.youtube.com/watch?v=Vk9bPDaZELQ`, 2019. (Accessed on 05/09/2019).

[53] G. Dulac-Arnold, N. Levine, D. J. Mankowitz, J. Li, C. Paduraru, S. Gowal, and T. Hester. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning*, 110(9):2419–2468, 2021.

[54] EasyList. EasyList. https://easylist.to/, 2022. Archived at https://perma.cc/T7S2-TZKH. (Accessed on 01/21/2022).

[55] EasyPrivacy. EasyPrivacy. https://easylist.to/easylist/easyprivacy.txt. (Accessed on 01/21/2022).

[56] S. Elmalaki. Fair-iot: Fairness-aware human-in-the-loop reinforcement learning for harnessing human variability in personalized iot. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, pages 119–132, Virtual, May 2021. ACM.

[57] S. Elmalaki, H.-R. Tsai, and M. Srivastava. Sentio: Driver-in-the-loop forward collision warning using multisample reinforcement learning. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 28–40, Shenzhen, China, Nov. 2018. ACM.

[58] ExoClick. The innovative ad company. https://www.exoclick.com/, 2020. (Accessed on 07/26/2020).

[59] eyeo. Snippet filters tutorial — adblock plus help center. https://help.eyeo.com/adblockplus/snippet-filters-tutorial. (Accessed on 06/10/2020).

[60] V. Gabillon, M. Ghavamzadeh, and A. Lazaric. Best arm identification: A unified approach to fixed budget and fixed confidence. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, pages 3212–3220, Lake Tahoe, Nevada, Dec. 2012. Curran Associates Inc.

[61] M. Garcia. Circumvention of ad blockers? not on our watch. – eyeo gmbh. https://eyeo.com/circumvention-of-ad-blockers-not-on-our-watch/, September 2018. (Accessed on 05/04/2020).

[62] A. Gleave, M. Dennis, C. Wild, N. Kant, S. Levine, and S. Russell. Adversarial policies: Attacking deep reinforcement learning. In *International Conference on Learning Representations*, Virtual, Apr. 2020. ICLR.

[63] Google. The privacy sandbox: Technology for a more private web. https://privacysandbox.com/, 2023. (Accessed on 07/19/2023).

[64] Google Chrome. chrome.webrequest. https://developer.chrome.com/extensions/webRequest, 2020. (Accessed on 05/06/2020).

[65] greiner. Adblock plus • view topic - why anti-circumvention filter list not operated by easylist? https://adblockplus.org/forum/viewtopic.php?f=4&t=59473, September 2019. (Accessed on 05/23/2020).

[66] D. Gugelmann, M. Happe, B. Ager, and V. Lenders. An automated approach for complementing ad blockers' blacklists. In *Proceedings on Privacy Enhancing Technologies*, volume 2, pages 282–298, Philadelphia, PA, June 2015. De Gruyter Open.

[67] gwarser. Resources Library . `https://github.com/gorhill/uBlock/wiki/Resources-Library`, 2019. (Accessed on 05/09/2019).

[68] S. Heinecke and L. Reyzin. Crowdsourced pac learning under classification noise. In *Proceedings of the AAAI Conference on Human Computation and Crowdsourcing*, volume 7, pages 41–49, Skamania Lodge, WA, Oct. 2019. AAAI Press.

[69] hfiguiere. #6969 (implement abort-on-property-read snippet) – adblock plus issue tracker. `https://issues.adblockplus.org/ticket/6969`, March 2019. (Accessed on 06/11/2020).

[70] IAB Tech Lab. Ad blocking detection script and improved user experience are keys to a better value exchange. `https://iabtechlab.com/standards/ad-blocking/deal/`. (Accessed on 05/04/2020).

[71] N. Immorlica, K. A. Sankararaman, R. Schapire, and A. Slivkins. Adversarial bandits with knapsacks. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 202–219, Baltimore, MD, Nov. 2019. IEEE.

[72] U. Iqbal, Z. Shafiq, and Z. Qian. The ad wars: Retrospective measurement and analysis of anti-adblock filter lists. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pages 171–183, New York, NY, USA, 2017. ACM.

[73] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq. Adgraph: A graph-based approach to ad and tracker blocking. In *IEEE Symposium on Security and Privacy (SP)*, pages 763–776, San Francisco, CA, May 2020. IEEE.

[74] M. Jethani. Adblock plus and (a little) more: Adblock plus 3.3 for chrome, firefox and opera released. `https://adblockplus.org/releases/adblock-plus-33-for-chrome-firefox-and-opera-released`, August 2018. (Accessed on 05/04/2020).

[75] X. Jia, X. Wei, X. Cao, and H. Foroosh. Comdefend: An efficient image compression model to defend adversarial examples. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6077–6085, Los Alamitos, CA, June 2019. IEEE Computer Society.

[76] S. Katariya, L. Jain, N. Sengupta, J. Evans, and R. Nowak. Adaptive sampling for coarse ranking. In *International Conference on Artificial Intelligence and Statistics*, pages 1839–1848, Playa Blanca, Lanzarote, Canary Islands, Apr. 2018. PMLR.

[77] R. Kleinberg. Nearly tight bounds for the continuum-armed bandit problem. In L. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems*, volume 17, pages 697–704, Vancouver, Canada, Dec. 2004. MIT Press.

[78] R. Kleinberg and T. Leighton. The value of knowing a demand curve: Bounds on regret for online posted-price auctions. In *44th Annual IEEE Symposium on Foundations of Computer Science*, pages 594–605, Cambridge, MA, Oct. 2003. IEEE.

[79] Kromtech Alliance Corp. Stopad for tv. https://stopad.io/tv, 2019.

[80] L. Kudryavtseva. New ad-tech terms: “ad reinsertion”, “ad recovery”, “ad replacement”. https://adguard.com/en/blog/ad-reinsertion.html, March 2017. (Accessed on 03/18/2020).

[81] I. T. Lab. Standards. https://iabtechlab.com/standards/, 2023. (Accessed on 07/19/2023).

[82] H. Le. CV-Inspector: Towards Automating Detection of Adblock Circumvention: Project Overview. https://athinagroup.eng.uci.edu/projects/cv-inspector/, January 2021. (Accessed on 01/04/2021).

[83] H. Le. AutoFR Project Page. https://athinagroup.eng.uci.edu/projects/ats-on-the-web/, 2023. (Accessed on 01/05/2023).

[84] H. Le, S. Elmalaki, A. Markopoulou, and Z. Shafiq. AutoFR: Automated Filter Rule Generation for Adblocking. In *32nd USENIX Security Symposium (USENIX Security)*, Anaheim, CA, Aug. 2023. USENIX Association.

[85] H. Le, A. Markoupoulou, and Z. Shafiq. CV-Inspector: Towards automating detection of adblock circumvention. In *The Network and Distributed System Security Symposium (NDSS)*, Virtual, Feb. 2021. The Internet Society.

[86] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *The Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019. The Internet Society.

[87] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th International Conference on World Wide Web*, pages 661–670, Raleigh, NC, Apr. 2010. ACM.

[88] Y. Li, Z. Yang, Y. Guo, and X. Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26, Buenos Aires, Argentina, May 2017. ACM.

[89] A. Locatelli, M. Gutzeit, and A. Carpentier. An optimal algorithm for the thresholding bandit problem. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1690–1698, New York, NY, June 2016. PMLR.

[90] N. Lomas. Adblock Plus maker has a new taskforce to fight publisher efforts to reinject ads. https://techcrunch.com/2018/09/19/adblock-plus-maker-has-a-new-taskforce-to-fight-publisher-efforts-to-reinject-ads/, 2018. (Accessed on 05/09/2019).

[91] MDN. Text: Web APIs. https://developer.mozilla.org/en-US/docs/Web/API/Text, 2022. (Accessed on 01/06/2022).

[92] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl. Block me if you can: A large-scale study of tracker-blocking tools. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 319–333. IEEE, 2017.

[93] H. Minhas. Project moonshot: Experimentation with machine learning based ad blocking. https://www.youtube.com/watch?v=1nJfvtvOOs0, 2022. Archived at https://perma.cc/CMJ7-QJLR. (Accessed on 01/28/2022).

[94] mjethani. Implement basic support for snippet filters . https://issues.adblockplus.org/ticket/6781, 2018. (Accessed on 05/09/2019).

[95] MoaAB: Mother of All AD-BLOCKING. https://forum.xda-developers.com/showthread.php?t=1916098, 2019.

[96] Mobiad. Home page. http://mobiadhome.com/, 2020. (Accessed on 07/26/2020).

[97] mozdev. Adblocker. https://web.archive.org/web/20021206021438/http://adblock.mozdev.org/, 2002. (Accessed on 01/28/2022).

[98] Mozilla. Mutationobserver - web apis. https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver, January 2020. (Accessed on 05/06/2020).

[99] Mozilla. Background CSS: Cascading style sheets. https://developer.mozilla.org/en-US/docs/Web/CSS/background, 2022. (Accessed on 01/31/2022).

[100] M. H. Mughees, Z. Qian, and Z. Shafiq. Detecting anti ad-blockers in the wild. *Proceedings on Privacy Enhancing Technologies*, 2017(3):130–146, 2017.

[101] B. Muthukadan. Selenium with python — selenium python bindings 2 documentation. https://selenium-python.readthedocs.io/, 2018. (Accessed on 05/09/2020).

[102] R. Nithyanand, S. Khattak, M. Javed, N. Vallina-Rodriguez, M. Falahrastegar, J. E. Powles, E. De Cristofaro, H. Haddadi, and S. J. Murdoch. Adblocking and counter blocking: A slice of the arms race. In *6th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 16)*, 2016.

[103] A. Oehler. How the platform works – help center. https://support.instart.com/hc/en-us/articles/220929867, October 2019. (Accessed on 03/18/2020).

[104] Optimizely. Above the fold. https://www.optimizely.com/optimization-glossary/above-the-fold/. (Accessed on 07/14/2020).

[105] Oracle. Data cloud registry. https://datacloudoptout.oracle.com/, 2023. (Accessed on 07/20/2023).

[106] Oriel. How it works. https://oriel.io/index.html#howitworks, 2020. (Accessed on 03/18/2020).

[107] Page Fair. The State of the Blocked Web. https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf, 2017. (Accessed on 05/09/2019).

[108] Pi-hole. Network-wide ad blocking. https://pi-hole.net/. (Accessed on 01/03/2022).

[109] Pi-hole. Customising Sources for Ad Lists. https://github.com/pi-hole/pi-hole/wiki/Customising-Sources-for-Ad-Lists, 2019.

[110] pkalinnikov. Issue 2449913002: Support websocket in webrequest api. - code review. https://codereview.chromium.org/2449913002/, 2017. (Accessed on 05/04/2020).

[111] V. L. Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156*, 2018.

[112] Publica. Products. https://dev.getpublica.com/products/, 2020. (Accessed on 07/27/2020).

[113] E. Pujol, O. Hohlfeld, and A. Feldmann. Annoyed users: Ads and ad-block usage in the wild. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, page 93–106, New York, NY, USA, 2015. Association for Computing Machinery.

[114] A. Rakhlin and K. Sridharan. Bistro: An efficient relaxation-based method for contextual bandits. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 1977–1985, New York, NY, June 2016. PMLR.

[115] ReviveAds. Ad reinsertion: An overview. http://news.reviveads.com/white-paper-reviveads-ad-reinsertion/. (Accessed on 03/18/2020).

[116] ReviveAds. Adblock circumvention strategies: Ad reinsertion, ad replacement, ad recovery. http://news.reviveads.com/adblock-circumvention-strategies/. (Accessed on 03/18/2020).

[117] K. Rogers. Why doesn't my ad blocker block 'please turn off your ad blocker' popups? - vice. https://www.vice.com/en_us/article/j5zk8y/why-your-ad-blocker-doesnt-block-those-please-turn-off-your-ad-blocker-popups, December 2018. (Accessed on 05/04/2020).

[118] sashachu. D: #8471 · abp-filters/abp-filters-anti-cv@d36effc. https://github.com/abp-filters/abp-filters-anti-cv/commit/d36effc62ec5207f5a6730127372a6cd3ebd1717, December 2018. (Accessed on 06/11/2020).

[119] scikit-learn. sklearn.ensemble.isolationforestdocumentation. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html, September 2016. (Accessed on 07/07/2020).

[120] scrapinghub. Python parser for adblock plus filters. https://github.com/scrapinghub/adblockparser, 2016. Archived at https://perma.cc/DN46-678C. (Accessed on 01/07/2022).

[121] A. Shuba, A. Markopoulou, and Z. Shafiq. NoMoAds: Effective and efficient cross-app mobile ad-blocking. In *Proceedings on Privacy Enhancing Technologies*, volume 4, pages 125–140, Barcelona, Spain, July 2018. Sciendo.

[122] S. Siby, U. Iqbal, S. Englehardt, Z. Shafiq, and C. Troncoso. WebGraph: Capturing advertising and tracking information flows for robust blocking. In *31st USENIX Security Symposium (USENIX Security)*, Boston, MA, Aug. 2022. USENIX Association.

[123] S. Singh. LEAN - IAB Tech Lab. https://iabtechlab.com/standards/ad-blocking/lean/, 2019. (Accessed on 05/09/2019).

[124] A. Sjösten, P. Snyder, A. Pastor, P. Papadopoulos, and B. Livshits. Filter list generation for underserved regions. In *Proceedings of The Web Conference 2020*, pages 1682–1692, Taipei, Taiwan, Apr. 2020. ACM.

[125] P. Snyder, A. Vastel, and B. Livshits. Who filters the filters: Understanding the growth, usefulness and efficiency of crowdsourced ad blocking. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, volume 4, Virtual, June 2020. ACM.

[126] SourcePoint. Homepage - sourcepoint. https://www.sourcepoint.com/, 2020. (Accessed on 03/18/2020).

[127] G. Storey, D. Reisman, J. R. Mayer, and A. Narayanan. The future of ad blocking: An analytical framework and new techniques. *CoRR*, abs/1705.08568, 2017.

[128] R. Sutton and A. Barto. *Reinforcement learning: an introduction.* The MIT Press, Cambridge, Massachusetts London, England, 2018.

[129] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *The Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2010. The Internet Society.

[130] F. Tramèr, P. Dupré, G. Rusak, G. Pellegrino, and D. Boneh. Adversarial: Perceptual ad blocking meets adversarial machine learning. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2005–2021, London, UK, Nov. 2019. ACM.

[131] Tranco. Information on the tranco list with ID XV9N. https://tranco-list.eu/list/XV9N/full, 2022. Archived at https://perma.cc/V76V-9JS2. (Accessed on 01/31/2022).

[132] R. Trimananda, H. Le, H. Cui, J. Tran Ho, A. Shuba, and A. Markopoulou. OVRseen: Auditing network traffic and privacy policies in oculus vr. In *31st USENIX Security Symposium (USENIX Security)*, Boston, MA, Aug. 2022. USENIX Association.

[133] uBlock Origin. Resources for uBlock Origin, uMatrix: static filter lists, ready-to-use rulesets, etc. . https://github.com/uBlockOrigin/uAssets, 2019. (Accessed on 05/09/2019).

[134] uBlock Origin. Getadmiral domains. https://raw.githubusercontent.com/LanikSJ/ubo-filters/master/filters/getadmiral-domains.txt, March 2020. (Accessed on 06/21/2020).

[135] uBlock Origin. gorhill/ublock: ublock origin - an efficient blocker for chromium and firefox. fast and lean. https://github.com/gorhill/uBlock, July 2020. (Accessed on 07/22/2020).

[136] E. Union. General data protection regulation (gdpr). https://gdpr-info.eu/, May 2018. (Accessed on 07/19/2023).

[137] J. Varmarken, H. Le, A. Shuba, A. Markopoulou, and Z. Shafiq. The tv is smart and full of trackers: Measuring smart tv advertising and tracking. In *Proceedings on Privacy Enhancing Technologies*, volume 2, pages 129–154, Virtual, July 2020. Sciendo.

[138] R. J. Walls, E. D. Kilmer, N. Lageman, and P. D. McDaniel. Measuring the impact and perception of acceptable advertisements. In *Proceedings of the Internet Measurement Conference*, pages 107–120, Tokyo, Japan, 2015. ACM.

[139] R. J. Walls, E. D. Kilmer, N. Lageman, and P. D. McDaniel. Measuring the impact and perception of acceptable advertisements. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, page 107–120, New York, NY, USA, 2015. Association for Computing Machinery.

[140] WaLLy3K. The Big Blocklist Collection. https://firebog.net, 2019.

[141] J. Wang, C. Song, and H. Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *The Network and Distributed System Security Symposium (NDSS)*, Virtual, Feb. 2021. The Internet Society.

[142] W. Wang, Y. Zheng, X. Xing, Y. Kwon, X. Zhang, and P. Eugster. Webranz: Web page randomization for better advertisement delivery and web-bot prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 205–216, New York, NY, USA, 2016. Association for Computing Machinery.

[143] wizmak. D: #9056 · abp-filters/abp-filters-anti-cv@ddd0c3d. https://github.com/abp-filters/abp-filters-anti-cv/commit/ddd0c3d9cd729d589519c57ba9aaa07229bdf10c, November 2018. (Accessed on 06/11/2020).

[144] Y. Xu, B. Kumar, and J. D. Abernethy. Observation-free attacks on stochastic bandits. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 22550–22561, Virtual, Dec. 2021. Curran Associates, Inc.

[145] Yandex. Yandex advertising network and ad exchanges. `https://yandex.com/support/direct/general/yan.html`, 2020. (Accessed on 07/26/2020).

[146] Z. Yang, W. Pei, M. Chen, and C. Yue. Wtagraph: Web tracking and advertising detection using graph neural networks. In *IEEE Symposium on Security and Privacy (SP)*, pages 1540–1557, San Francisco, CA, May 2022. IEEE.

[147] L. Yu, W. Xie, D. Xie, Y. Zou, D. Zhang, Z. Sun, L. Zhang, Y. Zhang, and T. Jiang. Deep reinforcement learning for smart home energy management. *IEEE Internet of Things Journal*, 7(4):2751–2762, 2019.

[148] J. Zhang, K. Psounis, M. Haroon, and Z. Shafiq. HARPO: Learning to subvert online behavioral advertising. In *The Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2022. The Internet Society.

[149] S. Zhu, X. Hu, Z. Qian, Z. Shafiq, and H. Yin. Measuring and disrupting anti-adblockers using differential execution analysis. In *The Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018. The Internet Society.