AutoFR: Automated Filter Rule Generation for Adblocking

Hieu Le[*] Salma Elmalaki[*] Athina Markopoulou[*] Zubair Shafiq[†]

[*]University of California, Irvine [†]University of California, Davis

Abstract

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. In this paper, 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 generates filter rules that can block 86% of the ads, as compared to 87% by EasyList, while achieving comparable visual breakage. Furthermore, AutoFR generates filter rules that generalize well to new sites. We envision that AutoFR can assist the adblocking community in filter rule generation at scale.

1 Introduction

Adblocking is widely used today to improve the security, privacy, performance, and browsing experience of web users. Twenty years after the introduction of the first adblocker in 2002, the number of web users who use some form of adblocking now exceeds 42% [6]. Adblocking primarily relies on filter lists (*e.g.,* EasyList [16]) that are manually curated based on crowd-sourced user feedback by a small community of filter list (FL) authors. There are hundreds of different adblocking filter lists that target different platforms and geographic regions [7]. It is well-known that the filter list curation process is slow and error-prone [3], and requires significant continuous effort by the filter list community to keep them up-to-date [31].

The research community is actively working on machine learning (ML) approaches to assist with filter rule generation [8, 21, 42] or to build models to replace filter lists altogether [1, 24, 41, 53]. 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 breakage. 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.

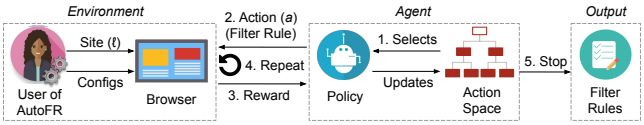
Fig. 1(a) illustrates the workflow of a FL author for creating rules for a particular site: (1) select a network request to block; (2) design a filter rule that corresponds to this request and apply it on the site; (3) visually inspect the page to evaluate if the filter rule blocks ads and/or causes breakage and; (4) repeat for other network requests and rules; since modern sites are highly dynamic, and often more so in response to adblocking [3, 14, 31, 55], 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. 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 [44]) 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

(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 filter 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 filter 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 more 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 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.

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 deploy and evaluate AutoFR's efficient implementation on Top–10K websites, and we find that the filter rules generated by AutoFR block 86% of the ads. We also find that they generalize well to new sites, *e.g.,* blocking 80% of the ads on the Top 5K–10K sites. The effectiveness of the AutoFR rules is overall comparable to EasyList in terms of blocking ads and visual breakage. Thus, we envision that the adblocking community will use AutoFR to automatically generate and update filter rules at scale.

The rest of our paper is organized as follows. Sec. 2 provides background and related work. Sec. 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 presents our implementation of the AutoFR framework. Sec. 5 provides its evaluation on the Top–10K sites. Sec. 6 concludes the paper.

2 Background & Related Work

Filter Rules. Adblockers have relied on filter lists since their inception. 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 [37]. There are different types of filter rules. The most popular type is *URL-based filter rules*, which block network requests to provide performance and privacy benefits [43]. Other types of filter rules are element-hiding rules (hide HTML elements) and JS-based rules (stop JS execution). Filter rules can also be per-site (*i.e.,* they are only allowed to trigger for particular sites) or treated as global rules (*i.e.,* allowed to trigger for any sites). Popular filter lists, such as EasyList, support these rules. Per-site rules are denoted with the "$domain" option in EasyList. This paper focuses on URL-based, per-site rules.

Filter Lists and their Curation. 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 [16] 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.

The research community has looked into the filter list curation process to investigate its effectiveness and pain-points [3, 31, 43, 50]. Snyder *et al.* [43] 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. 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.* [3] 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). 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. 3.1.

Machine Learning for Adblocking. Motivated by these challenges, prior work has explored using machine learning (ML) to assist with filter list curation or replace it altogether.

One line of prior work aims to develop ML models to automatically generate filter rules for blocking ads [8,21,42]. Bhagavatula *et al.* [8] trained supervised ML classifiers to detect advertising URLs. Similarly, Gugelmann *et al.* [21] trained supervised ML classifiers to detect advertising and tracking domains. Sjosten *et al.* [42] is the closest related to our work. First, they trained a hybrid perceptual and web execution classifier to detect ad images [10]. 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 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 [31]. Furthermore, the design of these rules did not automatically consider potential breakage.

Another line of prior work, instead of generating filter rules, trains ML models to automatically detect and block

ads [1, 2, 24, 41, 44, 53]. AdGraph [24], WebGraph [41], and WTAGraph [53] represent web page execution information as a graph and then train classifiers to detect advertising resources. Ad Highlighter [44], Sentinel [2], and PERCIVAL [1] 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 [36]. On the reliability front, concerns about the accuracy and brittleness of ML models in the wild [1, 2, 42], combined with a lack of explainability [46], 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.

ML-assisted FL Curation. There is, however, optimism in using ML-based approaches to assist with *maintenance* of filter lists. For example, Brave [42], Adblock Plus [2], and the research community [31] 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).

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. 3. Within the vast literature in RL [45], we choose the Multi-Arm Bandits (MAB) framework [4], for reasons explained in Sec. 3.2. Identifying the top–k arms [11, 35] rather than searching for the one best arm [19] has been used in the problems of coarse ranking [26] and crowd-sourcing [12, 22]. Contextual MAB has been used to create user profiles to personalize ads and news [33]. Bandits where arms have similar expected rewards, commonly called Lipschitz bandits [27], have also been utilized in ad auctions and dynamic pricing problems [28]. 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 [51]), as discussed in Sec. 3.3. While RL algorithms, in general, have been applied to several application domains [9, 17, 18, 54], RL often faces challenges in the real-world [15] including convergence and adversarial settings [5, 20, 23, 38, 52].

Our Work in Perspective. The design of the framework is described in Sec. 3 and illustrated in Fig. 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. Within the general 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.

3 AutoFR Framework

We formalize the problem of filter rule generation, including the process followed by human FL authors (Sec. 3.1 and Fig. 1(a)), our formulation as a reinforcement learning problem (Sec. 3.2 and Fig. 1(b)), and our multi-arm bandit algorithm for solving it (Sec. 3.3 and Alg. 1).

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 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

	Description	Filter Rule
1	eSLD	||*ad.com*^
2	FQDN	||*img.ad.com*^
3	With Path	||*ad.com/banners/* or ||*img.ad.com/banners/*

Table 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).

rule generation, revealed by prior work [3,31], 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. 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 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 filter 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 then try a more granular filter rule (*e.g.,* eSLD $\rightarrow$ FQDN from Table 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. 1(b) and formalized in the next section.

3.2 Reinforcement Learning Formulation

As described earlier and illustrated in Fig. 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. 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.

More 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 [4].

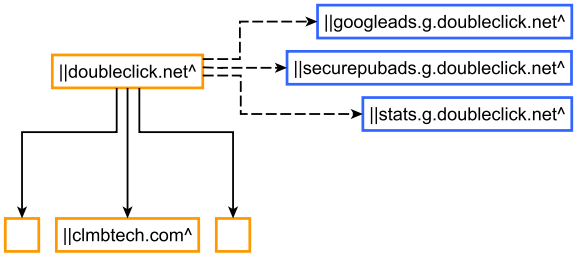


Figure 2: **Hierarchical Action Space.** A node (filter rule) within the action space has two different edges (*i.e.,* dependencies to other rules): (1) the initiator edge, $\rightarrow$, denotes that the source node initiated requests to the target node; and (2) the finer-grain edge, $\dashrightarrow$, targets a request more specifically, as discussed in Task 4 and Table 1.

The MAB framework fits well with our problem. The *MAB agent* replaces the human (FL author) in Fig. 1(a). The agent knows all available "arms" (possible filter rules), *i.e.,* the action space; see Sec. 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. 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. In 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. 3.2.2. Next, we define the key components of the proposed AutoFR framework, depicted in Fig. 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. 3.2.2, without requiring a human's visual inspection.

3.2.1 Actions

Action *a* (Filter Rule). An *action* is a URL blocking filter rule that can have different granular levels, shown in Table 1, and is applied by the agent onto the environment. We use the terms action, arm, and filter rule, interchangeably.

Hierarchical Action Space $\mathcal{A}_H$. 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. 2 shows an example of dependencies among candidate rules:

1. We should try rules that are coarser grain first (*doubleclick.net*) before trying more finer-grain rules (*stats.g.doubleclick.net*) (the horizontal dotted lines). 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. 4.2 describes how we retrieve the initiator information.

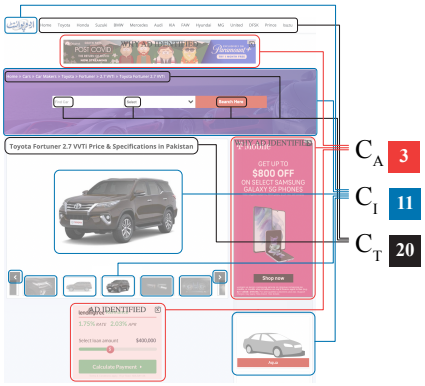


Figure 3: **Site Representation.** We represent a site as counts of visible ads (C_A), images (C_I), and text (C_T), as explained in Sec. 3.2.2. Applying a filter rule changes them, by blocking ads (reducing C_A) and/or hiding legitimate content (changing C_I and C_T, thus breakage $\mathcal{B}$).

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. 3.2.2), the agent no longer needs to explore its children. The creation of $\mathcal{A}_H$ automates Task 2.

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. 3. 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 [44]) 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. 4.3.

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:

$$\widehat{C}_A = \frac{\overline{C}_A - C_A}{\overline{C}_A}, \qquad \widehat{C}_I = \frac{|\overline{C}_I - C_I|}{\overline{C}_I}, \qquad \widehat{C}_T = \frac{|\overline{C}_T - C_T|}{\overline{C}_T} \quad (1)$$

$\widehat{C}_A$ 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. 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:

$$\mathcal{B} = \frac{\widehat{C}_I + \widehat{C}_T}{2} \quad (2)$$

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. 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 $\mathcal{R}_F$. 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. (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:

$$\mathcal{R}_F(w,\widehat{C}_A,\mathcal{B}) = \begin{cases} -1 & \text{if } \widehat{C}_A = 0 & \text{(3a)} \\ 0 & \text{if } \widehat{C}_A > 0\ ,\ 1-\mathcal{B} < w & \text{(3b)} \\ \widehat{C}_A & \text{if } \widehat{C}_A > 0\ ,\ 1-\mathcal{B} \geq w & \text{(3c)} \end{cases}$$

The rationale for this design is as follows.

a) *Bad Rules* (Eq. (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. (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. (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 ($\widehat{C}_A$).

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 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 [4]. 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.

3.3 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. In the end, it outputs a set of filter rules $\mathcal{F}$, if any. It consists of the two procedures discussed next.

[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. (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).

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 ($\varepsilon = 0.05$)
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 ($\mathcal{F}$)

procedure INITIALIZE(ℓ, n)
  $\overline{C}_A, \overline{C}_I, \overline{C}_T, reqs \leftarrow$ VISITSITE($\ell, n, \emptyset$)
  $\mathcal{A}_H \leftarrow$ BUILDACTIONSPACE($reqs$)
  return $\overline{C}_A, \overline{C}_I, \overline{C}_T, \mathcal{A}_H$
end procedure

procedure AUTOFR(ℓ, w, c, α, n)
  $\overline{C}_A, \overline{C}_I, \overline{C}_T, \mathcal{A}_H \leftarrow$ INITIALIZE(ℓ, n)
  $\mathcal{F} \leftarrow \emptyset$, $\mathcal{A} \leftarrow \emptyset$
  $\mathcal{A} \leftarrow \mathcal{A}_H$.root.children
  repeat
    $Q(a) \leftarrow Q_0, \forall a \in \mathcal{A}$
    for $t = 1$ to T **do**
      $a_t \leftarrow$ CHOOSEARMUCB($\mathcal{A}, Q_t, c$)
      $C_{A_t}, C_{I_t}, C_{T_t}, hits \leftarrow$ VISITSITE($\ell, 1, a_t$)
      $\widehat{C}_{A_t}, \widehat{C}_{I_t}, \widehat{C}_{T_t} \leftarrow$ SITEFEEDBACK($C_{A_t}, C_{I_t}, C_{T_t}$)
      $\mathcal{B}_t \leftarrow$ BREAKAGE($\widehat{C}_{I_t}, \widehat{C}_{T_t}$)
      if $a_t \in hits$ **then**
        $r_t \leftarrow \mathcal{R}_F(w, \widehat{C}_{A_t}, \mathcal{B}_t)$
        $Q_{t+1}(a_t) \leftarrow Q_t(a_t) + \alpha(r_t - Q_t(a_t))$
      else
        Put a_t to sleep
      end if
    end for
    $\mathcal{A} \leftarrow \{a.\text{children}, \forall a \in \mathcal{A} \mid -\varepsilon <= Q(a) <= \varepsilon\}$
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{\forall a \in \mathcal{A} \mid Q(a) > \varepsilon\}$
  until $\mathcal{A}$ is $\emptyset$
  return $\mathcal{F}$
end procedure

INITIALIZE Procedure. First, we obtain the baseline representation of a site of interest ℓ (Sec. 3.2.2), when no filter rules are applied. To do so, 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 set of outgoing *reqs*. The average counters will be used in evaluating the reward function (Eq. (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. 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

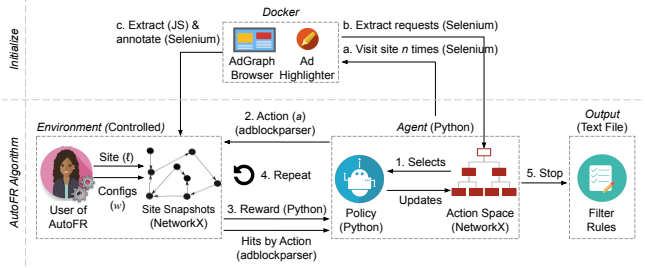


Figure 4: **AutoFR Example Workflow (Controlled Environment).** INITIALIZE (a–c, Alg. 1): (a) spawns $n = 10$ docker instances and visits the site until it finishes loading; (b) extracts the outgoing requests from all visits and builds the action space; (c) extracts the raw graph and annotates it to denote C_A, C_I, and C_T, using JS and Selenium. Once all 10 site snapshots are annotated, we run the RL portion of the 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*.

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. (3) as a guide. A technicality is that Eq. (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 ($\varepsilon = 0.05$) to decide if $Q_t(a)$ is close enough to zero ($-\varepsilon \leq Q(a) \leq \varepsilon$). Then, we apply the same intuition as in Eq. (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. (3a). If a rule is $Q(a) < \varepsilon$, then we ignore it and do not explore its children.
b) *Potentially Good Rules: Explore Further.* Mirroring Eq. (3b), if a rule is within a range of $\pm\,\varepsilon$ 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 on Fig. 2. 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 ($Q(a) > \varepsilon$), we add that rule to our list $\mathcal{F}$ and no longer explore its children. This mimicks Eq. (3c). An example is shown in Fig. 2: 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]. This stopping condition automates Task 5. The output is the final set of good filter rules $\mathcal{F}$.

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

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. 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 [30]).

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. 3.3. In this section, we present the design of this implementation in a controlled environment: AutoFR-C, or AutoFR for simplicity. An overview of our implementation is provided in Fig. 4. Importantly, this allows our AutoFR tool to scale across thousands of sites and, thus, utilized as a practical tool.

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 (*i.e.,* it can replicate our results and utilize snapshots in its own work).

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, steps a–c. We use AdGraph [24], 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. 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. 5. Site snapshots are large and contain thousands of nodes and edges. 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. 4.3), 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. 5. 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 [39], 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. 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. 5: (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 [31, 55]. 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 [31]) and try to evade it (*i.e.,* trigger different JS code), thus leading to a different response that is not captured by our snapshots. Another limitation can be explained via Fig. 5. 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.

4.2 Agent

Action Space $\mathcal{A}_H$. During the INITIALIZE procedure (Alg. 1), we visit the site ℓ multiple times and construct the action space from all the visits. First, we convert every request to three different filter rules, as shown in Table 1. We add edges between them (eSLD $\rightarrow$ FQDN $\rightarrow$ With path), which serve as the finer-grain edges, shown in Fig. 2. 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. 2. This makes the $\mathcal{A}_H$ taller and reduces the

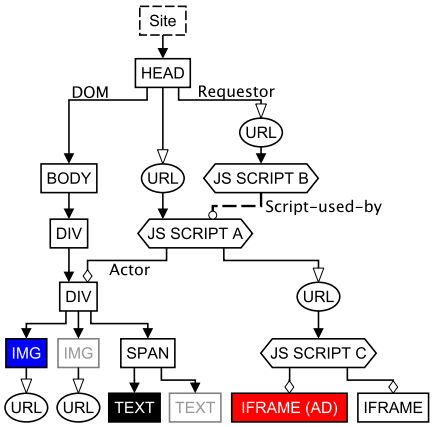


Figure 5: **Site Snapshot.** It is a graph that represents how a site is loaded. The nodes represent JS Scripts, HTML nodes (*e.g.,* DIV, IMG, TEXT, IFRAME), and network requests (*e.g.,* URL). "Actor" edges track which source node added or modified a target node. "Requestor" edges denote which nodes 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. 4.1, nodes annotated by AutoFR have filled backgrounds, while grayed-out nodes are invisible to the user.

number of arms to explore per run of MAB, as described in Sec. 3.3. The resulting action space is a directed acyclic graph with nodes that represent filter rules; Fig. 2 provides a zoom-in example. We implement it as a NetworkX graph and save it as a ".graphml" file, a standard graph file type utilized by prior work [42].

Policy. The UCB policy of Sec. 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.

4.3 Automating Visual Component Detection

A particularly time-consuming step in the human workflow is Task 3 in Fig. 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. 1(b) summarizes this assessment in the reward in Eq. (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 [44], a perceptual ad identifier (and web extension) that detects ads on a site. We evaluated different ad perceptual classifiers, including Percival [1], 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.

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 2: **AutoFR Top–5K Results.**

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 [31].

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 [1]. Text can be extracted from screenshots of a site using Tesseract [44], 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 detect visually, and assessing the success of a rule that blocks tracking is more challenging, *e.g.,* involves JS code analysis [14]. Extending AutoFR for tracking is a direction for future use.

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. In addition, we characterize properties of the filter rules produced by AutoFR: how they can be controlled via parameter w, how they compare to EasyList rules, how fast they need to be updated, and how well they generalize across sites.

5.1 Filter Rule Evaluation Per-Site

We apply AutoFR on the Tranco Top–5K sites [32,47] to generate rules using the breakage tolerance threshold of $w=0.9$. All other AutoFR parameters are the same as in Alg. 1.

AutoFR Results. Table 2 summarizes our results. 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, which is an order of magnitude improvement over the 13 hours per-site of live training in Sec. 4. During each per-site run, we explore tens to hundreds of potential rules and conduct up to thousands of iterations within MAB runs. This efficiency is key to scaling AutoFR to a large number of sites and over time.

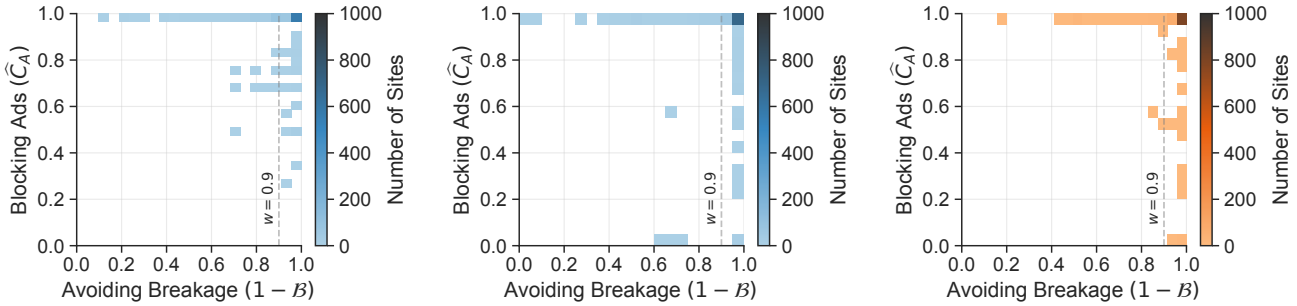
AutoFR: Validation with Snapshots. Since AutoFR generates rules for each particular site (*i.e.,* per-site), we first apply these rules to the site for which they have been created. To that end, we first apply the rules to the stored site snapshots, and we report the results in Fig. 6(a) and Table 3 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. 4.1) and the rules are more effective when tested on sites in the wild.

AutoFR *vs.* EasyList: Validation In The Wild. Next, we apply the rules from AutoFR to the same sites they have been created for, but this time on the real site ("in the wild"), not on the site snapshots. For comparison, we also apply EasyList[4] to the same set of Top–5K sites and we report our results in Fig. 6(b) and Table 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 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 [24].

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 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 observed two sites 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.

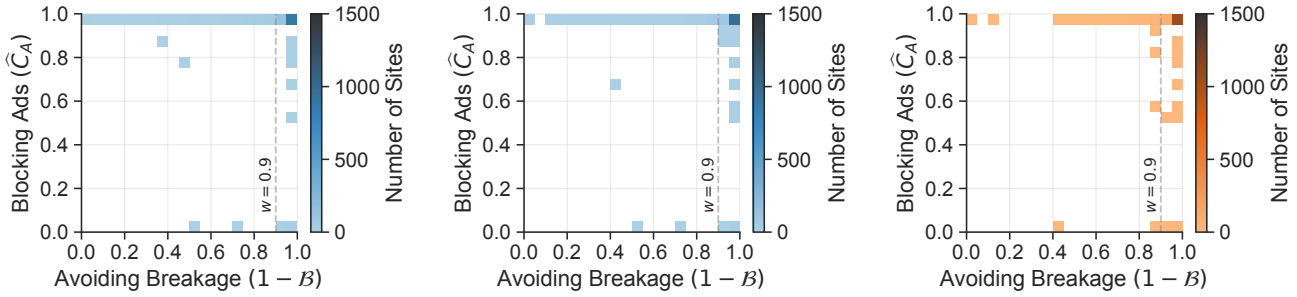
Tuning AutoFR via Threshold w. AutoFR is the first approach that can be tuned per-site and explicitly allows to express a preference. The FL author that uses AutoFR must select the site to create rules for and express their preference by tuning a knob (threshold w).

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



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

Figure 6: **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 3, col. 1, 2, and 4, for additional information.

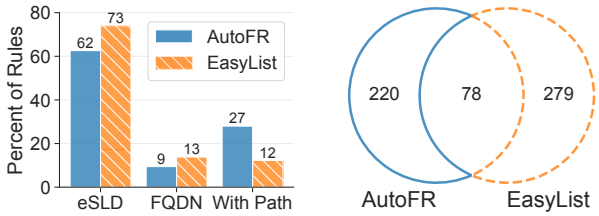


(a) AutoFR (All Rules) (b) AutoFR (Rules from ≥ 3 sites) (c) EasyList (In the Wild)

Figure 7: **Testing Filter Rules on New Sites (Top 5K–10K, In the Wild).** We create two filter lists, Fig. 7(a) with all rules from *W09-Dataset* and Fig. 7(b) that contains rules that were created for ≥ 3 sites. We test them in the wild on the Top–5K to 10K sites (new sites) and show their effectiveness along with EasyList (Fig. 7(c)). We observe that Fig. 7(b) performs better, blocking 8% more ads than Fig. 7(a). Table 3, col. 6–8, contains additional information.

		Sec. 5.1, Fig. 6, Top–5K				Sec. 5.3.1	Sec. 5.3.3, Fig. 7, Top–5K to 10K		
		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)*	***AutoFR*** *(All rules) (In the Wild)*	***AutoFR*** *(≥ 3 sites) (In the Wild)*	***EasyList*** *(In the Wild)*
	Description ($w=0.9$)	1	2	3	4	5	6	7	8
1	Sites in operating point: $\widehat{C}_A \geq 0.95, 1-\mathcal{B} \geq 0.95$	62%	74%	85%	79%	72%	67%	73%	80%
2	Sites within w: $\widehat{C}_A > 0, 1-\mathcal{B} \geq 0.9$	77%	**86%**	**85%**	87%	82%	76%	**80%**	87%
3	Ads blocked within w: $\sum_\ell (\overline{C}_A \times \widehat{C}_A) \,/\, \sum_\ell \overline{C}_A$; $1-B \geq 0.9$	70%	**86%**	**84%**	87%	78%	72%	**80%**	86%

Table 3: **Results.** We provide additional results to Fig. 6 and 7, within their respective sections. 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. 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. 5.3.1), we repeat the same experiment of col. 2 during July 2022 for a longitudinal study of AutoFR rules.



(a) Rule Types (b) Grouped by eSLD

Figure 8: **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.

5.2 AutoFR *vs.* EasyList: Comparing Rules

We compare the rules generated per-site by AutoFR and EasyList from Sec. 5.1. For a fair comparison, we only consider EasyList rules that are *triggered* when visiting sites.

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. 8(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. 8(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 different rules based on FQDNs like ||*d2na2p72vtqyok.cloudfront.net*^. 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 particular 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. 4.1 that AutoFR generates rules with a conservative approach when using site snapshots, and thus will consider finer-grain rules for some cases to avoid breakage. Whereas FL authors manually 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, for site *pastemagazine.com*, AutoFR has ||*pastemagazine.com/common/js/ads-gam-a9-ow.js*, while EasyList has *pastemagazine.com/common/js/ads-*. Partial paths within EasyList may extend the life of a filter rule over time for some sites. We further evaluate this in Sec. 5.3.1. AutoFR can extend to partial paths in the future.

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. 8(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. 5.1.

Findings. Depicted in Fig. 8(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 challenging 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. (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

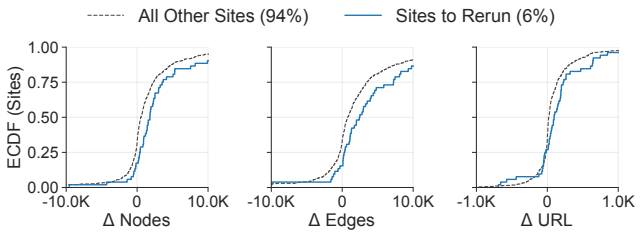


Figure 9: **Δ Site Snapshots between July vs. January 2022.** The differences in site snapshots for nodes, edges, and URLs. A positive change in the x-axis denotes that July had more of the respective factor, while a zero denotes no change.

space when we find a good rule following the intuition from Sec. 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. 8(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. 5.1 and Table 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, amp-accordion-0.1.js for functionality). AutoFR balances the trade-off between blocked ads and breakage, see Sec. 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.

5.3 Robustness of AutoFR Filter Rules

AutoFR generates rules for a particular site and uses snapshots collected at a particular time. Next, we investigate and discuss how well these rules perform over time, across different sites, and in adversarial scenarios.

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. 5.1) to the same sites in July 2022 and summarize the results in Table 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. 9 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 $\pm$1K changes in nodes and edges, respectively; while 40% of sites have more than $\pm$100 changes in URL nodes. Compared to sites to rerun, 75% of sites have more than $\pm$1K changes in nodes and edges, while 65% of sites have more than $\pm$100 changes in URL nodes. As expected, the 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.

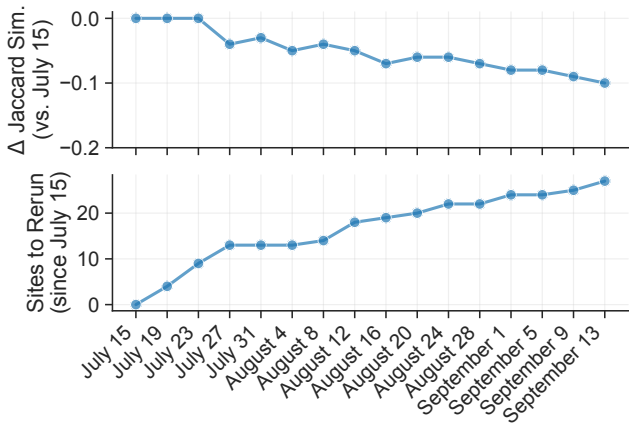


Figure 10: **Longitudinal Study Every Four Days.** We conduct a finer-grain longitudinal study of 100 sites over a two-month period. We find that over time, site snapshots will become less similar (*i.e.,* negative Δ Jaccard similarity), often denoting that rules may be less effective. FL authors can rerun AutoFR on these sites that change more frequently to output effective rules.

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. 5.3.1). Fig. 10 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. 5.3.1. As time passes, the similarity between site snapshots will naturally decrease, which denotes 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 only 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 like ||*charlotteobserver.com*/.../0*a*086549941921*c*9*ac*8*e.js*. 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 one month apart (*e.g., liputan*6*.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.

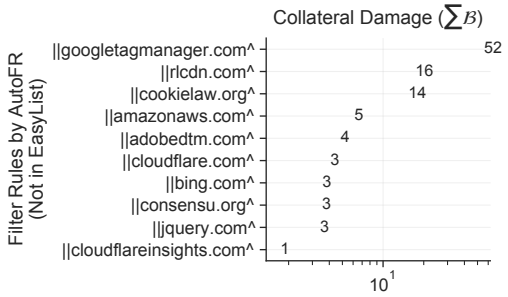


Figure 11: **Collateral Damage of Global Rules.** AutoFR rules are generated per-site and can potentially cause breakage when applied to other sites (*i.e.,* treated as a global rule). We report the rules that are unique to AutoFR (*i.e.,* not part of EasyList), ordered by decreasing total collateral 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.

5.3.3 From Per-Site Rules To Global Filter Lists

AutoFR generates URL-based filter rules for a particular site. Similarly, EasyList supports per-site rules as well. It currently contains ∼800 per-site rules. Although these rules are guaranteed to perform well on the sites that they have been designed for (as demonstrated in Sec. 5.1), it is not guaranteed that the same rules are as effective when applied to other sites, *i.e.,* used as "global" rules.

Collateral Damage. In Fig. 11, we report the potential collateral damage, defined as the sum of breakage ($\sum\mathcal{B}$), caused when AutoFR rules are treated as global rules. Rules are considered global when applied to sites other than the ones they have been created for. We observe that they tend to block tag managers (*e.g.,* ||*googletagmanager.com*^, ||*adobedtm.com*^), CDNs or cloud storage services (*e.g.,* ||*cloudflare.com*^, ||*amazonaws.com*^, ||*rlcdn.com*^), third-party libraries (*e.g.,* ||*jquery.com*^), and cookie consent forms (*e.g.,* ||*cookiekaw.org*^, ||*consensu.org*^). These rules target domains that can serve legitimate content and ads across dif-

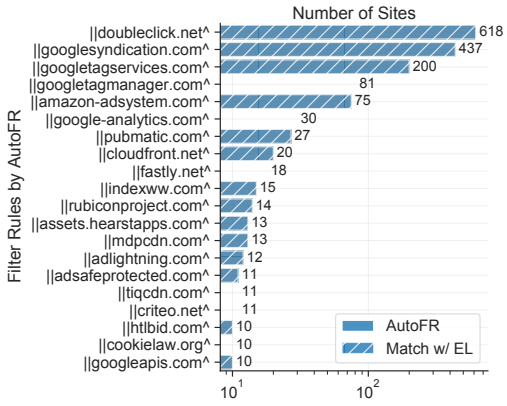


Figure 12: **Top–20 Filter Rules by AutoFR for Top–5K Sites.** They include the main advertising and tracking services, such as Alphabet (*doubleclick.net*), Amazon (*amazon-adsystem.com*), and PubMatic (*pubmatic.com*). Thus, they are likely to generalize well.

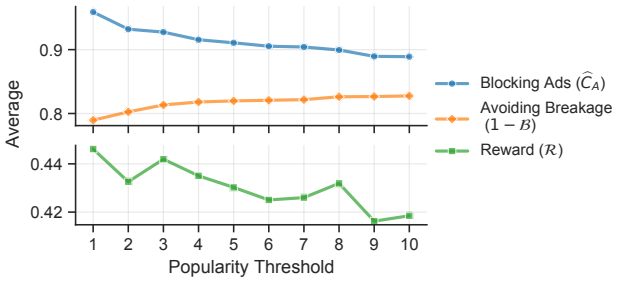


Figure 13: **Selecting Per-Site Rules into Global Filter Lists.** After creating the per-site AutoFR rules for each site (with $w = 0.9$), we create 10 global filter lists. "Popularity 1" means that a rule is selected into the global list if it was generated in at least one site; "popularity 10" means 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.

ferent 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 [3].

Towards Global Filter Lists. 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. We denote this as the "popularity" of a rule. Fig. 12 shows the Top–20 AutoFR most common rules across sites. They intuitively make sense as they belong to widely used advertising and tracking services. Therefore, we utilize these heuristics as criteria to select AutoFR rules to include in filter lists. Once selected, we now treat them as global rules. As the popularity increases, the global filter list contains fewer global rules, resulting in fewer blocked ads but less breakage. We show the results in Fig. 13.

We analyze in detail two global filter lists. First, "popularity 1" 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 ≥ 3 sites. Fig. 13 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. (3)). We apply these global filter lists on the Tranco Top 5K–10K sites in the wild. Fig. 7 and Table 3 col. 5–6 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. 7(b) compares relatively well against Fig. 7(c) (EasyList): 73% of sites are in the desired operating point (top-right corner), *vs.* 80% by EasyList (row 1, col. 7–8). Overall, the rules generated from the Top–5K sites were able to block 80% of ads on the Top 5K–10K sites. This shows good generalization of AutoFR rules across unseen sites, which agrees with Fig. 12.

5.3.4 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 [31]. 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 [31, 46]. 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 [46]. 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 Ad Highlighter with a better future visual perception tool, or pre-process the logos to remove adversarial perturbations [25]. At the system level, as an adversarial bandits problem, where the reward received from pulling an arm comes from an adversary [5].

6 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. Thus, we envision that AutoFR will be used by the adblocking community to automatically generate and update filter rules at scale. An extended version of this paper, including appendices, can be found at [30].

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. AutoFR can also be applied 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 [40, 48, 49]. On mobile and smart TVs specifically, one could leverage existing tools to automatically explore apps or mobile browsers [13, 34, 40, 49].

Availability. The AutoFR implementation, generated filter rules, and the dataset are available at [29].

Acknowledgments

This work is supported in part by the National Science Foundation under award numbers 1956393, 1900654, 1815666, 2051592, 2102347, 2103038, 2103439, 2105084, and 2138139. We would like to thank the USENIX Security reviewers for their feedback, which helped to improve the paper. We would like to thank Stelios Stavroulakis for his help during the early stages of this work. Lastly, special thanks to the filter list community, including Ryan Brown, Arthur Kawa, and Peter Lowe, who provided valuable insight into the human process of creating and maintaining filter rules.

References

[1] 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, July 2020.

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

[3] 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.

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

[5] 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.

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

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

[8] 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. ACM, Nov. 2014.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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, May 2021.

[15] 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.

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

[17] 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. ACM, May 2021.

[18] 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.

[19] 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.

[20] 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*, Apr. 2020.

[21] 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.

[22] 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.

[23] 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.

[24] 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, May 2020.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] H. Le, S. Elmalaki, A. Markopoulou, and Z. Shafiq. AutoFR: Automated filter rule generation for adblocking (extended version). *arXiv:2202.12872*, 2022.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

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

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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. ACM, June 2020.

[44] 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.

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

[46] 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.

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

[48] 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.

[49] 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. Sciendo, July 2020.

[50] 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.

[51] 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)*, Feb. 2021.

[52] 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. Curran Associates, Inc., Dec. 2021.

[53] 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.

[54] 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.

[55] 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.