Michael Smith, Peter Snyder, Moritz Haller, Benjamin Livshits, Deian Stefan, and Hamed Haddadi

Blocked or Broken? Automatically Detecting When Privacy Interventions Break Websites

Abstract: A core problem in the development and maintenance of crowdsourced filter lists is that their maintainers cannot confidently predict whether (and where) a new filter list rule will break websites. The enormity of the Web prevents filter list authors from broadly understanding the compatibility impact of a new blocking rule before shipping it to millions of users. This severely limits the benefits of filter-list-based content blocking: filter lists are both overly conservative (i.e. rules are tailored narrowly to reduce the risk of breaking things) and error-prone (i.e. blocking tools still break large numbers of sites). To scale to the size and scope of the Web, filter list authors need something better than the current status quo of user reports and manual review, to stop breakage before it has a chance to make it to end users.

In this work, we design and implement the first automated system for predicting when a filter list rule breaks a website. We build a classifier, trained on a dataset generated by a combination of compatibility data extracted from the EasyList filter project and novel browser instrumentation, and find that our classifier is accurate to practical levels (AUC 0.88). Our open-source system requires no human interaction when assessing the compatibility risk of a proposed privacy intervention. We also present the 40 page behaviors that most predict breakage in observed websites.

Keywords: content blocking, Web compatibility

DOI Editor to enter DOI
Received ..; revised ..; accepted ...

Michael Smith: University of California, San Diego (mds009@eng.ucsd.edu)
Peter Snyder: Brave Software (pes@brave.com)
Moritz Haller: Brave Software (mhaller@brave.com)
Benjamin Livshits: Imperial College London (b.livshits@imperial.ac.uk)
Deian Stefan: University of California, San Diego (deian@cs.ucsd.edu)
Hamed Haddadi: Brave Software (hamed@brave.com)

1 Introduction

A large and growing body of research has shown that filter-list-based content blocking significantly improves Web privacy[1, 2], security[3, 4], and performance[5, 6]. The proliferation of cosmetic-only rules in popular filter lists suggests that filter lists significantly improve the user-perceived aesthetics of Web browsing. And the ever-expanding popularity of extensions and browsers that include filter-list-based content blocking suggests that filter lists are important to large swaths of Web users.

While the benefits of filter lists are well studied and understood, systematizing and automating the creation of these lists remains an open challenge. This is largely because research is very good at measuring the *benefits* of blocking network requests (e.g. number of trackers blocked, data saved, CPU cycles reduced), but comparatively poor at measuring the *costs* of blocking requests (e.g. number of websites broken or user-desirable features impacted). In effect, Web researchers mainly count one side of the ledger, and as a result, filter list curation in practice remains a nearly completely manual process, consisting of activists and community members making best-effort predictions of the Web-scale impact of filter list rules. The result is that filter lists are both too conservative (i.e. there are things that filter list authors would like to block, but don't to avoid breaking sites) and too liberal (i.e. content blocking tools still break plenty of websites).

Additional human labor will not fundamentally improve the situation. Because of the size and constantly-changing nature of the Web, any efforts by filter list authors to *manually* evaluate the Web-wide impact of a filter list rule will be incomplete, and dramatically so. As a result, users of filter list tools end up being both the consumers and testers of new filter rules. This means broken sites for users, and in some cases giving up on the privacy, security, and performance wins of content blocking tools.

1.1 Problem Difficulty

We need an automated way to predict the Web compatibility impact of a new or updated filter list rule, so

that rules can be tested, tailored, and optimized before being shipped to users. Alas, this is a tough problem for several reasons.

First, determining if a page is broken is difficult because "brokenness" presents itself in a variety of forms. A page can be "broken" in an obvious way (e.g. the page is blank), in a very subtle way (e.g. a form on a deeply-nested page does not submit correctly), and everything in between. Furthermore, compatibility breakage sometimes only reveals itself after a user interacts with a page or attempts to trigger some interactivity.

Another obstacle to automated detection is the difficulty of assembling a large dataset of "broken" web sites. Both site authors and filter list maintainers have strong incentives to fix broken sites as quickly as possible (and, generally, with as few people noticing as possible). This makes it harder for researchers to obtain a generalized understanding of the problem, and so makes developing automated detection systems tricky.

1.2 Contributions

This work improves the state of filter list content blocking by designing a fully automated classifier that accurately predicts whether a filter list rule breaks a website, in the subjective evaluation of a browser user. Our classifier requires no human interaction to run and takes advantage of deep browser engine instrumentation, and so can scale far beyond what is possible with human assessments. Our classification pipeline takes as input i) a filter list rule and ii) a Web page URL, and returns a prediction of whether executing the given Web page *with the given filter list rule applied* will break the page.

We build our classification dataset in two novel steps. First, we use the commit history of the EasyList filter project to build up a labeled dataset of Web page URLs paired with filter list rules that cause either a breaking or non-breaking change when applied to the page. Second, we use a heavily modified version of a Chromium-based browser to analyze the execution of these Web pages with and without the corresponding rules. Significantly, our modified browser records both what events occurred during the Web page's execution (e.g. which scripts were executed, which DOM nodes were inserted or modified, which event listeners were registered), and which actors on the page were responsible for each event (e.g. which script fetched a given resource, or inserted a DOM element, or fetched a dependent script). Our instrumented browser then allows us to export the recording of each page execution as an XML-encoded directed graph.

We combine these sources of data to generate a large dataset of Web page executions paired with filter rules labeled as causing either a breaking or non-breaking page behavior change. We then extract 433 features from the execution recordings in each of these samples, and train a classifier that performs with AUC of 0.88 to predict sample labels.

More specifically, this work offers the following contributions:

1. The design of a multi-step **fully automated system** for accurately predicting whether a privacy intervention (i.e. a filter list rule) would break a website, in the subjective evaluation of a browser user.
2. A **public dataset** consisting of 1,469 unique real-world filter list rules, applied to 2,570 unique Web pages that they affect, resulting in 2,662 recordings of page behavior changes, each labeled with whether the applying the rules yielded a broken or working version of the page.
3. A detailed **discussion of which page behaviors predicted pages breaking** (and which page behaviors did not).
4. The **open source implementation**[1] of both our data collection pipeline and classifier, implemented in a Chromium-based browser and scikit-learn.

2 Motivation and Overview

2.1 A Brief Introduction to Filter Lists

Filter lists are collections of regular-expression-like rules describing trust statements over URLs. The most common applications of filter lists are in browsers and browser extensions to block unwanted requests when browsing the Web (e.g. requests for trackers, unwanted advertisements, distracting page content, etc). Usually filter list rules describe origins and paths that should be blocked, but most tools that apply filter lists have additional syntax to further restrict how and when each rule should be applied. For example, rules can be restricted to only be applied to certain kinds of requests (e.g. images, sub-documents, scripts) or only applied in certain

1 https://github.com/brave-experiments/webcompat-measurement-pipeline

contexts (e.g. specifying that some rules should only be considered when visiting certain sites).

Most popular filter lists are crowd-sourced by communities that add and refine rules in large shared lists. Rules are added when a list contributor finds out about a new tracking script (or otherwise unwanted Web resource) and decides to block it. For example, assume a filter list maintainer is browsing a site and notices the site has included a tracking script, served from https://tracker.example/bad.js. The filter list maintainer, wanting to protect other users, adds a new rule to the filter list, instructing the browser to block the tracker from loading on the current site[2]. The filter list maintainer then tests out the rule by revisiting the site with the new rule applied. The maintainer sees that the tracking script is now blocked, and that the site continues working correctly. Having checked that the filter list rule works (i.e. the target script was blocked) and that the site still works, the filter list maintainer commits the new rule to the filter list, which is soon downloaded by millions of filter list subscribers, blocking the tracking script on that specific site for all users.

2.2 Breaking Sites is Too Easy

Filter list maintainers, though, have to choose between privacy and compatibility. Worse, they often have to try and choose between these goals without data, relying only on intuition and best guesses.

To see why, refer back to the example discussed in the previous sub-section. The filter list author specified that the tracking script should only be blocked when it is included by one specific site; the tracking script will continue to be loaded on every other site on the web, continuing to harm users despite the filter list author identifying the script as a tracker. The privacy harm continues because the rule was written narrowly.

Alternatively, the filter list author could have written the rule to be general, and to block the tracking script whenever it was included on *any* site[3]. This would prevent more privacy harm, but risks breaking sites. The filter list author only checked that one specific site still worked when the script was blocked; other sites might have integrated the script in such a way that they break if the script is not present. This is common, and happens when pages rely on utility functions tracking scripts provide or otherwise deeply integrate the tracking script.

In the scenario where the filter list author commits the general rule, not only has the author broken an unknown number of pages but, worse, the author won't find out about the breakage until *after the rule has been shipped to users*, when users start encountering broken sites and (hopefully) reporting issues. The underlying problem is that filter list maintainers have no scaleable, automated way to test rules before shipping them. Maintainers can browse sites with the rule enabled, but this only works for rules tied to a small number of sites: it does not scale to real rules that impact huge fractions of the Web. The Web is too large, and the number of filter list maintainers too small.

2.3 Towards Automated Detection of Breakage

Filter list authors need tools to help them protect user privacy, while minimizing risks to compatibility. The ideal solution would be an oracle that allowed filter list authors to submit a proposed filter list rule and receive back a list of sites the rule would break. This system would be automated so that filter list authors can repeatedly and quickly query it, allowing rules to be optimized (i.e. maximizing privacy while minimising breakage) before shipping them to users.

In practice this is difficult. Determining if a site is broken is tricky for a variety of reasons. The broken functionality might not be immediately obvious, and might only be triggered after interacting with the page. Blocking a script on one site might not affect its users at all, while blocking the same script on another site might break the site entirely. Breaking a page might not have any visual side effect, only manifesting itself through unintended application flow. These are just some examples of why "site breakage" is a difficult classification problem.

However, as a step towards building an automated site breakage oracle, we designed a system that predicts whether a site will break, given three inputs: i) a Web page (described by its URL), ii) a filter list rule, and, optionally, iii) a browser profile, allowing the browser to be arbitrarily configured before classification. We developed our system using a ground truth dataset constructed from the commit history of the EasyList project, and consisting of tuples of i) a Web page URL, ii) a filter list rule, and iii) whether the filter list rule broke the site (Section 3.1). We visited each URL in a

2 This rule might look like `||tracker.example/bad.js$domain=site.example`.

3 This general rule might look like `||tracker.example/bad.js`.

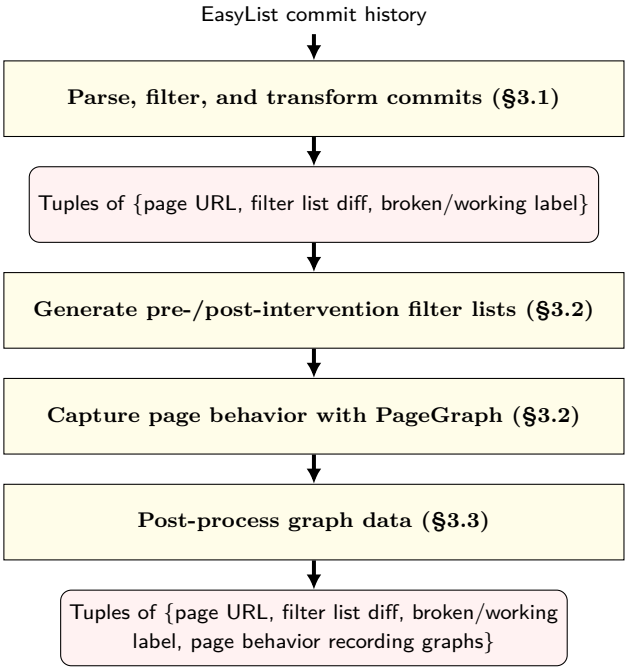

Fig. 1. Pipeline diagram of our Web compatibility dataset generation process.

crawler instrumented to record the page's execution at an extremely detailed level (Section 3.2). We then extracted 433 features from the execution record for each site, and used those features to train a classifier (Section 4.3) that performs with mean AUC of 0.88 (Section 5.1). Finally, we used the classifier to learn 40 features that predict whether a filter list rule will break a website (Section 5.2).

3 Dataset

Our first contribution is a dataset of examples of filter list changes and their effects on page behavior, labeled with whether or not those effects represent Web compatibility breakage from a user perspective, and the novel methodology with which we assembled this dataset at a sufficiently large scale for ML classifier training. The dataset contains a total of 2,662 examples, each consisting of a page URL, a corresponding filter list change affecting the page, a broken-or-working label, and recordings of how the page's behavior responds to the change. Figure 1 summarizes our data collection pipeline.

```
P: https://www.mealty.ru/catalog/ (Fixes
  https://forums.lanik.us/viewtopic.php?t=47335)
---

easyprivacy/easyprivacy_allowlist_international.txt:
  ...
  @@||mc.yandex.ru/metrika/tag.js$script,
    domain=auto.yandex|coddyschool.com
+ @@||mealty.ru/js/ga_events.js$~third-party
  @@||megafon.ru/static/?files=*/tealeaf.js
  ...
```

Fig. 2. A sample Web compatibility fix commit excerpted from the EasyList repository[4], inserting an exception rule to allow through a script which shares its filename with a popular analytics script. Blocking the script breaks a page on `mealty.ru`.

```
A: https://tinyzonetv.to/
Block adserver at https://tinyzonetv.to/
---

easylist/easylist_adservers.txt:
  ...
  ||sftapi.com^
+ ||sfzover.com^
  ||sg2rgnza7k9t.com^
  ...
```

Fig. 3. A sample coverage-expanding commit excerpted from the EasyList repository[5], inserting a rule to block an ad server at `sfzover.com`, found on the site `tinyzonetv.to`.

3.1 Collecting Examples of Broken and Working Sites

To train our classifier to detect when a filter list change breaks a site, we first needed a set of examples for the classifier to learn from, of sites breaking when such a change is introduced. Manually hunting through the Web for broken sites, and then debugging filter lists to identify the rules responsible in each instance, would have been too time- and labor-intensive given the dataset size required to effectively train and test the classifier: our final dataset contains over a thousand such examples. Moreover, this approach would place us as the judges of page brokenness, a subjective measure: our judgments may differ from those of end users.

We sidestep these problems by mining labeled examples of Web page breakage from a non-traditional source: the commit logs of the EasyList project[6], a large and widely-used community-maintained filter list distribution. The EasyList authors use the Git version con-

4 https://github.com/easylist/easylist/commit/a509c21b72c2d4959bff0539408282 1f207730fd
5 https://github.com/easylist/easylist/commit/0c453dbe0882640ce16dc823fc72dc3aaa55ec62
6 https://easylist.to/

trol system to coordinate the development of their filter lists, so each update to the rules is logged with an associated commit message, numbering over 169,000 across the project's history. The commit messages follow uniform conventions agreed on by the authors. In particular, a rule update to fix Web compatibility breakage should be tagged with the prefix "P:" and reference the URL of at least one page on which the problem occurs. Figure 2 shows a sample compatibility-fix commit taken from EasyList. These commits are often made in response to user reports, and are further vetted by the domain-expert maintainers that merge them into the EasyList repository; therefore, we claim that they represent something close to ground truth for page breakage as perceived by end users. Each of these commits typically comprises one or a few rule additions and/or deletions, constituting a filter list change which repairs compatibility breakage on the referenced page. Inverting the change—i.e., flipping additions to deletions and deletions to additions—produces a filter list change which *breaks* the page instead of repairing it. By scanning the EasyList commit history for Web compatibility commits, parsing out the associated URLs, and applying this inversion to each commit, we seeded our dataset generation with *positive* examples of filter list changes that introduce breakage, tied to specific Web pages on which that breakage occurs.

In order to teach our classifier to distinguish filter list changes which break pages, we also needed to show it examples of changes which *don't* cause breakage. These changes should still have an effect on their target pages, but a desirable one: blocking an ad, for example, or circumventing a privacy-invading tracker. Again we returned to the EasyList commit logs, where such changes are tagged with the prefix "A:", and also reference page URLs on which the intended effects can be observed. Figure 3 shows a sample from the EasyList Git repository. We applied the same process of scanning the commit history, minus the inversion step, to seed our dataset generation with *negative* examples: non-breaking filter list changes and specific Web pages they affect.

3.2 Capturing Page Behavior

It would be difficult if not impossible to predict page breakage by looking just at Web page URLs and filter rule source code. Instead, our classifier draws its predictions from the way those pages behave at runtime in a browser equipped with content blocking, and how their behavior is altered by changes to the content blocker's

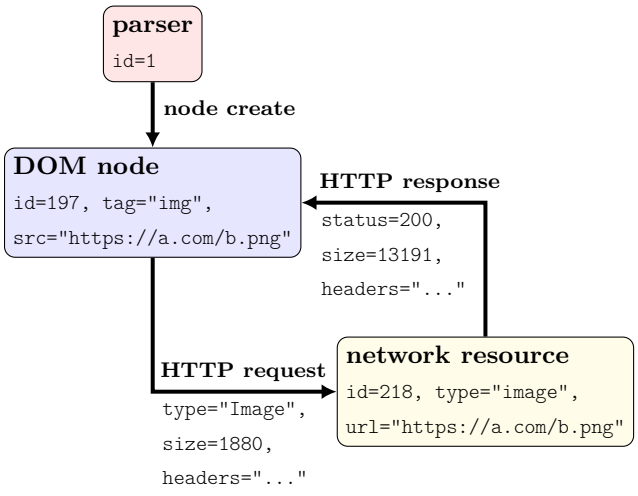

Fig. 4. PageGraph encoding of the initialization of an image element. The browser's HTML parser creates a DOM node to represent a decoded `<img>` tag. An HTTP request is dispatched to retrieve the image file pointed to by the image element's `src` attribute, and a success response is received.

filter lists. We recorded page behavior with PageGraph[7], our deep browser instrumentation system built into the Brave Browser. Every page opened in a PageGraph-enabled browser build is monitored at runtime, and its activity across the browser engine is recorded in a unified directed graph structure. Nodes of this graph correspond to interacting entities in the mini-ecosystem of a Web page: actors like scripts, the parser, and the content blocker which perform actions, and those that are acted upon, like network resources, DOM nodes, Web APIs, and filter rules. Edges represent the actions that connect them: `node insert` rules between the parser and DOM nodes, `resource block` edges between filter rules and network resources, `API call` edges between script actors and Web APIs; as well as `structure` edges which record parent-child relationships between DOM nodes. Figure 4 shows a sample excerpt from a PageGraph graph, illustrating how a Web request to load the image pointed to by the URL of an HTML `<img>` tag is encoded as a pair of `HTTP request` and `HTTP response` edges, connecting the DOM node for the `<img>` tag to the `network resource` node representing the image file at the given URL, all annotated with metadata captured from this runtime interaction.

From the collection of positive and negative examples assembled in Section 3.1, we used our PageGraph-

7 https://github.com/brave/brave-browser/wiki/PageGraph

enabled Brave Browser build to crawl each Web page twice, capturing its behavior both *with* and *without* the corresponding filter list change (the "intervention") applied. First, we used the EasyList Git repository to generate a version of the filter list without the change. We then injected this filter list into the browser's content blocker, replacing the built-in filter list. Our crawler launched the browser, controlling it with the Puppeteer automation framework[8], and navigated to the page URL. The crawler waited for the DOM `load` event to fire, and a further 15 seconds beyond that to give the page time to fully settle. The page's behavior during this time was captured by PageGraph and exported in graph form at the end of the browsing session, taking a baseline for the page's normal operation; we refer to this as a *pre-intervention* graph. Next, we generated a second version of the filter list, this time with the change applied, and injected it into the content blocker. Repeating the crawling process produced a second graph, representing the page's behavior under the influence of the filter list change; we refer to this as a *post-intervention* graph. For positive examples, this graph reflects the page in a broken state; for negative examples, it reflects a desirable, intentional change in behavior (e.g., blocking an ad).

Because we were using historical data—the EasyList commit log—to drive our crawling, there was a chance that the pages we were visiting had been altered or had even disappeared in the time since the original commits were made. As an initial heuristic, we only considered commits dated 2013 or later. We rejected any pages which produced an error response (e.g., 404 Not Found) during crawling. We further validated that applying the corresponding filter list change still actually had an effect on the page, by using the `adblock-rust`[9] library to evaluate the filter rules with and without the change against the page's recorded network activity. If there was no difference between the blocking decisions made in the two cases, the page was excluded from our dataset. At the completion of crawling and filtering, we were left with a dataset of 2,662 examples; later analysis (§5.3) would indicate that this is more than enough to train a useful classifier.

Another approach to accounting for the historical nature of the EasyList data could be to proxy crawler Web requests through cached versions of the target pages contemporary with the EasyList commit dates. We tried this approach using the Internet Archive's Wayback Machine[10] as that cache, but ultimately discarded the idea. Snapshots cached by the Archive proved to be of insufficient fidelity and completeness to accurately reproduce page behavior. For example, we observed that requests sent by ad-network-integration code were often missing from the cache, introducing errors that would not have occurred on live pages. Since the filtered dataset proved to be of sufficient size for our purposes, we did not explore this line of inquiry further in our study.

3.3 Post-Processing Data

At this point, we had two recorded PageGraph graphs for each example, one representing the page's behavior without the corresponding filter list change applied ("pre-intervention"), and one representing its behavior with the change in place ("post-intervention"). We post-processed this dataset to generate a third "intervention-only" graph per example, which approximated the delta in page behavior caused by the intervention. Each intervention-only graph is a sub-graph of the pre-intervention graph, containing graph nodes and edges describing the behavior of the parts of the page that would be blocked if the intervention were applied. We hypothesized that providing these narrowed-down sub-graphs as part of the input to our classifier would help it find a stronger signal, tuning out some of the noise of surrounding page behavior unaffected by the filter list change, while helping to control for the dynamism of Web content.

To generate an intervention-only graph, we first identify `network resource` nodes in the pre-intervention graph that the content blocker allowed through under the pre-intervention filter rules, but which it would have blocked under the post-intervention ruleset. These nodes represent the network resources covered by the filter rule change[11]. Starting from these resource nodes, we selectively walk outward in the pre-intervention graph, marking additional nodes and edges for inclusion in the sub-graph. We walk up from each resource node to the node which caused the resource

8 https://puppeteer.github.io/puppeteer/

9 https://github.com/brave/adblock-rust

10 https://web.archive.org

11 Note that we identify these `network resource` nodes by applying the post-intervention filter rules to network traffic in the pre-intervention condition, and not by cross-referencing with the post-intervention graph. This avoids introducing noise from unrelated variations in page behavior between multiple visits to the page.

to be requested over the network, e.g., from image resource nodes to the HTML `<img>` DOM nodes pointing to those images; we mark these nodes and the connecting `HTTP request/response` edges for inclusion. For any HTML `<script>` DOM nodes we discover, we follow `script execute` edges leading out from them to `script actor` nodes that represent those scripts running in the browser's JavaScript engine. Finally, we mark all additional nodes which are reachable by taking one step from any already-marked node, as well as the connecting edges. This captures, e.g., the effects that scripts blocked by the filter list change have on the page when not blocked, like the insertion or modification of DOM nodes, the registration and un-registration of event listeners, and Web API calls. Extracting the marked nodes and edges produces a new graph focused on the effects of the intervention on page behavior.

4 Classifier Construction

We aim to construct a machine learning (ML) classifier with sufficient accuracy to predict whether a filter list rule will break a site. Additionally, we aim to analyze each classifier's predictions to better understand which features predict site breakage. As a simple baseline, we employed a classical feature-based ML model over an end-to-end learning system such as deep neural networks. The learning task is posed as a binary classification problem with the positive label *"site did break"* and the negative label *"site did not break"*.

In this sections we outline the data pre-processing and feature extraction, followed by a description of the full classification pipeline.

4.1 Data Pre-Processing

Our dataset numbers 2,662 examples, each consisting of a "pre-intervention" graph, a "post-intervention" graph, and an "intervention-only" graph (as defined in Section 3.3). We excluded any examples with empty intervention-only graphs (indicating no measured effect on the page resulting from the intervention). This left 1,966 training samples with 1,011 positive labels and 955 negative labels. As a pre-processing step, we converted each graph into pandas[12] data frames, with each graph represented as an edge list.

4.2 Generating Candidate Features

We next describe how we generated the set of candidate features considered when constructing our classifier (Section 4.3 describes how we determined which features had significant predictive value, and Section 5.2 presents which features ended up being predictive).

To generate the candidate features, we first defined three dimensions that might be useful for predicting site breakage; each dimension is described below. We then generated 433 features by selecting different options from each feature dimension.

4.2.1 Scope of Analysis

The first dimension we considered was whether to extract features from i) the behavior of the overall page, or ii) the behavior that was blocked by the the filter list rule.

Page scope features capture whether patterns in a pages' overall behavior predicts breakage. These features would be predictive if certain aspects of the page's design and implementation predicted breakage, independent of what was blocked on the page. For example, "page scope" features might detect if site complexity *in general* predicts breakage. We extracted "page scope" features from the "pre-intervention" graph (as described in Section 3.3).

Conversely, **intervention scope** features look for patterns that predict breakage specifically in the page behaviors blocked or modified by the filter list rule. These features will be predictive if what is being blocked predicts breakage (instead of the page context that blocking is occurring in). "Intervention scope" features would be predictive if, for example, blocking certain JavaScript API calls causes pages to break. We extracted "intervention scope" features from the "intervention-only" graph (see Section 3.3).

12 https://pandas.pydata.org/

4.2.2 Absolute vs. Relative Values

The second feature dimension we considered was whether to quantify behaviors using i) absolute counts, or ii) as relative ratios.

Features using **absolute counts** are based on the number of times an event or element was observed during a page's execution. For example, an "absolute count" feature would be based on the number of video elements that were embedded in a page, or the number of network calls that were blocked by a filter list. "Absolute count" features could be used to detect if the size of a page, or the number of images on a page, can predicted breakage, independent of how many elements or images were blocked.

Features targeting **relative counts**, on the other hand, capture what percentage of occurrences of an event on a page were blocked by the filter list rule. For example, a "relative count" feature would consider the percentage of images blocked on a page, or the number of network requests prevented because of the filter list rule.

4.2.3 Expertly Curated vs. Automatically Generated Target Behaviors

The third feature dimension we considered was what strategy we used to decide what page behaviors to measure. Some of the behaviors we targeted were manually curated, drawing from domain knowledge and "expert" intuition; other features were automatically generated by examining generic graph features.

The **expert curated** features were generated by extrapolating from our domain knowledge and past experience dealing with broken websites. Our domain knowledge comes form various sources, including our research and experiences contributing to and maintaining privacy tools[13]. We then generalized our observations about how websites break, and tried to capture those generalizations as features. Some examples of "expert curated" features we generated include i) whether a blocked script fetches additional scripts, or ii) whether a blocked script registered event handlers in the page.

We also **automatically generated** a large number of additional features that considered counts of different graph attributes (node types, edge types, node attributes, edge attributes). These automatically generated features did not consider the relationship between different graph elements, or the semantics of the values for node and edge attributes. As a result, the "automatically generated" features were generally much simpler than the kinds of page behaviors captured in the "expert curated" features. Some examples of "automatically generated" features include i) how many DOM nodes of each HTML tag appeared in the page (HTML tag names are recorded in node attributes in PageGraph), or ii) how many Web APIs were called during a page's execution (Web API invocations are recorded in PageGraph with edges of type "call").

We then categorized our features (whether "expert curated" or "automatically generated") into one of the following five categories, depending on the kinds of page behaviors were measured by the feature.

(see Section 5.2).

- **HTML structure**: aspects of the structure and composition of the document (e.g. numbers of different tags, amount of text on the page)
- **JavaScript modifications of page structure**: measures of how the page's structure was constructed or modified by scripts (e.g. numbers of DOM nodes inserted by scripts, amount of text modified by scripts)
- **Other JavaScript behaviors**: script operations and calls not directly related to modifying DOM structure (e.g. counts of API calls made by scripts, numbers of event handlers registered by scripts)
- **Network behaviors**: the sub-resources and other network calls made during page execution (e.g. number of bytes fetched, number of sub-resources fetched)
- **Generic graph features**: graph measurements with considering the behaviors being encoded by the graph (e.g. number of nodes and edges, number unique node attribute values)

We note that this categorization was not used in the training or evaluating of the classifier; this categorization was instead to benefit the later discussion of what kinds of page behaviors predicted breakage (Section 5.2).

4.3 Classification Pipeline

The classification pipeline consists of the model to learn the target function based on the extracted features as described in the previous section, as well as several steps to transform the inputs before making predictions.

13 *Links and references omitted for anonymization.*

We select XGBoost[14], a popular model choice which has been shown to achieve state-of-the-art performance across a wide range of prediction tasks [7]. As a tree-based method, XGBoost has several characteristics that make it particularly suitable to serve as an off-the-shelf baseline method. First, variable selection is performed automatically, making it immune to the inclusion of irrelevant features. We empirically verify this by conducting recursive feature selection over all 433 input features, and found that it has no significant effect on performance. Second, tree-based methods are robust to outliers due to the way they partition the input space. We verify this empirically by removing the top 1-percentile of training samples w.r.t. their node/edge ratio, finding no significant effect on performance. Last, tree-based methods naturally deal with missing values (see e.g. [8, 9]).

Despite the ability for tree-based methods to handle missing values and correlated features, we conduct the following transformations on the input data in order to improve robustness of the model and reduce training time:

1. All features which contain a percentage of empty values above a certain threshold are dropped (empirically we found a good threshold to be 0.85, see section 5.1).
2. All features which have a Pearson correlation coefficient above a certain threshold with at least one other feature, are dropped (empirically we found a good threshold to be 0.73, see section 5.1).
3. All remaining features are standardised by removing the mean and scaling to unit variance.

5 Classifier Evaluation

This section outlines the analysis of the classifier described in the previous section with respect to its predictive and explanatory power, as well as its sample complexity. We divide the evaluation into three parts:

1. In section 5.1 we train, tune and test the classifier in order to optimise its predictive performance and show whether it is possible to achieve practical utility on the PageGraph data set.
2. In section 5.2 we train and test the classifier without tuning on subsets of features to analyse the effect of individual features on the predictions.
3. In section 5.3 we train and test the classifier without tuning on training data samples of increasing size, including all features. The aim is to understand how much data is needed to achieve practical utility.

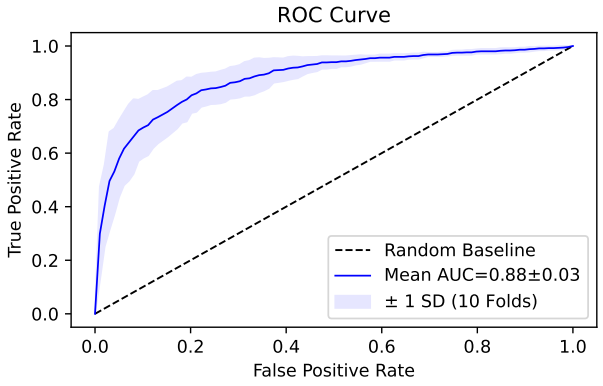

Fig. 5. ROC-AUC curves averaged over 10 folds with hyper-parameter tuning to evaluate the practical utility of the classifier.

To summarize and evaluate the accuracy of the predicted class probabilities, we use the area under the receiver operating characteristic curve (ROC-AUC) as a threshold-invariant metric.

5.1 Practical Utility

First, we evaluate the performance of the classifier with respect to its practical utility. To that end, we conduct several hyper-parameter optimization rounds to improve the predictive performance of the classifier. We use nested cross-validation to estimate the generalization error and prevent over-fitting. We only explore a small range of values around the parameters' default values. All hyper-parameters and their tuning ranges are given in Table 3 in the appendix. In the inner loop, the validation fold is rotated across 3 folds to choose the best hyper-parameter configuration (modelled as samples from a *Gaussian Process* using the framework of *Bayesian Optimization* with 10 parameter configurations being tested per fold). Hereby, we optimize over all pipeline steps including the thresholds for dropping null-valued and correlated features. The outer loop with 10 folds is used to evaluate the performance of the learner. The evaluation is conducted over all 1,966 training samples with all 433 features.

As shown in Figure 5, we find that the classifier can achieve practical utility, with an average ROC-AUC of 0.88($\pm$0.03) across the outer 10 folds. We list accuracy

14 https://github.com/dmlc/xgboost

Fold	Accuracy	ROC-AUC	AP
1	0.781726	0.834881	0.807511
2	0.796954	0.855301	0.864379
3	0.888325	0.932653	0.953064
4	0.837563	0.911097	0.930447
5	0.847716	0.909447	0.922484
6	0.837563	0.918936	0.920909
7	0.821429	0.893486	0.910913
8	0.785714	0.857426	0.889738
9	0.780612	0.888275	0.904671
10	0.724490	0.827410	0.860115

Table 1. Practical utility of the classifier (§5.1) as measured by Accuracy, ROC-AUC and Average-Precision (AP) scores across 10 folds with hyper-parameter tuning.

and average precision (AP) metrics across all folds in Table 1. We thus show that an off-the-shelf classifier can extract enough signal from the PageGraph data set to separate positive examples (breaking) from negative examples (non-breaking) well above random (i.e. a ROC-AUC of 0.50). Depending on the modelling objective it is possible to further optimize performance with respect to precision and recall or type I and type II error by tuning the classification threshold.

5.2 Feature Importance

Next, we analyze the explanatory power of the classifier by inspecting the effect of the extracted features on predictions. First we conduct the analysis along the lines of the feature generating dimensions by removing all features belonging to a specific dimension and measuring the drop in predictive performance. We then repeat the analysis for individual features to establish a rank-order of the most predictive features. Despite the fact that the *Leave-One-Covariate-Out*[10] importance metric doesn't capture interaction effects if applied to single features, it can nonetheless hint at the relative amount of signal contained in individual features.

To estimate the importance metric for feature groups and single features respectively, we first fit the classifier with default parameters (no tuning) and all 433 features to serve as a baseline. We then generate a new data set for each feature (group) by removing the respective features from the training data before fitting the model with the remaining features. We then estimate the importance by subtracting the ROC-AUC of the reduced feature set from the baseline for each feature respectively via 5-fold cross validation and report the mean and standard deviation.

The results for the importance of feature groups per feature generating dimension are shown in Figure 6. Page features in isolation result in a slightly higher loss than Intervention features, with a mean loss of ROC-AUC 0.010 and 0.004 respectively. More pronounced is the difference between expertly curated vs. automatically generated features with the auto-generated features incurring a mean loss of ROC-AUC 0.004 and 0.037 respectively. Slightly less pronounced is the difference between absolute and relative features with a mean loss of ROC-AUC of 0.024 and 0.009 respectively.

Results for the importance of individual features are shown in Table 2, and are described in detail in the following section.

5.2.1 Features with Predictive Power

This process yielded 40 features, from an initial starting set of 433 candidate features. Table 2 gives the 40 features that had predictive power in our model. A wide range of features ended up being useful, representing a wide range of page behaviors (e.g. JavaScript API calls, structure and size of the DOM, amount and size of network calls). As illustrated in Figure 6, no single kind of page behavior dominated the predictive power of the classifier (though, as discussed later, features capturing pages' network behaviors were somewhat more predictive than other categories of features).

Similarly, no other "dimenson" of features dominated the classifier's predictions. Nearly as many predictive features measured overall page behaviors (20) as measured just the activities blocked by the filter list rule (20). Similar numbers of expert-curated features were predictive as automatically generated features (15 and 25, respectively).

However, there were some trends we observed in which page behaviors predicted page breakage. We here briefly note three interesting and surprising patterns we observed.

First, the kinds and number of JavaScript features used on a page predicted page breakage. JavaScript behaviors that were blocked (i.e. the scripts that were blocked by the filter list rule) were generally more predictive than overall page behaviors (i.e. blocked and not blocked scripts alike). The number of scripts fetched and executed by blocked scripts, the number of cookies set by blocked scripts, and the num-

Rank	Scope	Source	AUC Loss	Description
Page Structure Features				
3	Intervention	Auto	0.00374	% of sub-document requests blocked
6	Page	Auto	0.00222	# of tags and text nodes in initial HTML
13	Intervention	Auto	0.00156	Δ in # of sub-documents after blocking
19	Page	Auto	0.00078	# of `<iframe>` in page
33	Page	Auto	0.00025	% of DOM nodes that are `<html>`
36	Page	Auto	0.00018	% of DOM nodes that are `<iframe>`
40	Intervention	Expert	0.00012	% of <html> elements blocked
Generic Graph Features				
11	Page	Auto	0.00179	# of unique node and edge types
22	Intervention	Auto	0.00061	# of unique types of actions taken by blocked scripts
35	Page	Auto	0.00023	# of unique types of actions in entire page
Features Regarding JavaScript Modifying Page Structure				
7	Intervention	Auto	0.00217	# of DOM nodes created by HTML parser prevented by blocking
12	Intervention	Auto	0.00169	% of JS DOM nodes created by blocked scripts
16	Intervention	Auto	0.00116	# of DOM node insertions done by blocked scripts
21	Intervention	Expert	0.00062	# of <html> elements created by blocked scripts
25	Intervention	Expert	0.00046	# of DOM nodes created by blocked scripts
30	Page	Auto	0.00032	# of DOM nodes created by scripts in entire page
34	Intervention	Auto	0.00024	% of DOM nodes deletions done by blocked scripts
Other JavaScript Features				
4	Page	Expert	0.00295	# of times any script accessed properties on `window.navigator`
5	Intervention	Auto	0.00254	# of scripts fetched or eval'ed by blocked scripts
8	Page	Expert	0.00216	# of times any script deleted a value from `sessionStorage`
9	Intervention	Expert	0.00205	% of `document.cookie` sets occurring in blocked scripts
10	Intervention	Auto	0.00190	% of `localStorage` operations occurring in blocked scripts
14	Page	Auto	0.00126	# of scripts fetched or eval'ed in entire page
15	Intervention	Expert	0.00120	# of times blocked scripts read from `document.cookie`
17	Page	Auto	0.00113	# of `document.cookie` operations in entire page
18	Intervention	Auto	0.00083	% of `sessionStorage` operations done by blocked scripts
20	Page	Expert	0.00073	# of WebGL calls, over the entire page
23	Intervention	Auto	0.00059	# of Web API calls made by blocked scripts
26	Intervention	Auto	0.00042	% of `eventListener` removals done by blocked scripts
28	Page	Auto	0.00034	# of `eventListener` registrations in entire page
29	Intervention	Expert	0.00033	% of `window.navigator` reads made by blocked scripts
31	Page	Auto	0.00027	# of <script> tags in page
37	Page	Expert	0.00017	# of `window.screen` reads over entire page
38	Page	Auto	0.00016	# of cross-document script-reads in entire page
39	Page	Expert	0.00013	# of `localStorage` reads over entire page
Network Features				
1	Intervention	Expert	0.00831	Δ in bytes sent over network after blocking
2	Intervention	Expert	0.00550	size of resources directly blocked
24	Intervention	Expert	0.00059	# of resources blocked (direct or indirect)
27	Page	Auto	0.00034	% of page actions that were network requests
32	Intervention	Expert	0.00026	% of network resources that were blocked

Table 2. This table presents the 40 features (from the 433 features considered) that predicted page breakage. The "Rank" column gives the relative importance of each feature, with 1 being the most predictive, 40 the least. "Scope" describes whether the feature was extracted from the *pre-intervention* graph, denoted "Page" or *intervention-only* graph, denoted "Intervention" (Section 4.2.1). "Source" gives whether the feature was developed through expert curation or automatic generation (re Section 4.2.3. "AUC Loss" gives how much predictive power was lost when the feature was removed (Section 4.3). "Description" provides a terse description of the page behavior captured by the feature.

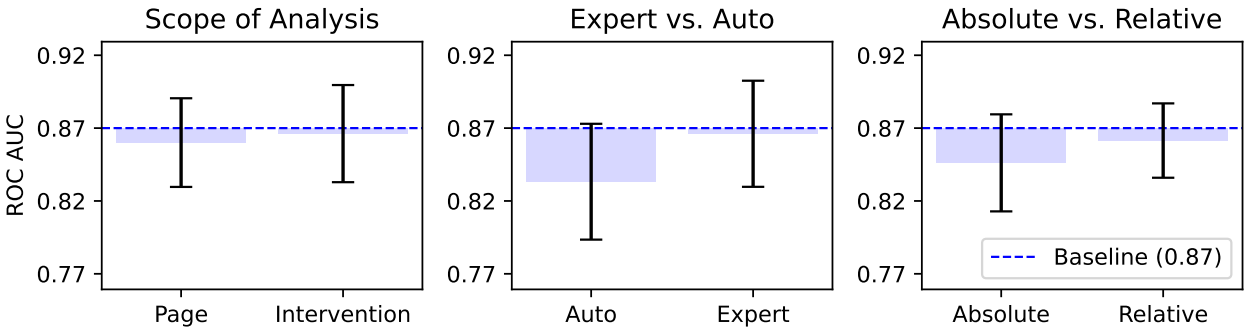

Fig. 6. Importance of the feature generating dimensions **Scope of Analysis**, **Absolute vs. Relative Values** and **Expertly Curated vs. Automatically Generated Target Behaviors** as measured by the mean loss in ROC-AUC when removing the respective features from the training data. Error bars indicate the standard deviation across 5 folds.

ber (and %) of DOM nodes injected by blocked scripts were all predictive of breakage.

These findings support an intuition that most Web applications are highly modular, with the privacy-threatening portions of each application being contained in a small portion of the overall implementation. However, this does not mean that the privacy-harming parts of Web applications can generally be cleanly severed from the overall application. In fact, these modules tend to be tightly coupled, and blocking privacy-harming scripts often breaks the overall application. This conclusion is supported by other related work[11] that finds that filter-list-based content blocking tools are insufficiently granular to effectively address many kinds of privacy harms on the Web.

Second, the number of sub-documents in a page predicts page breakage. Many of behaviors that predict breakage relate to the number of sub-documents included on the page, both directly (e.g. "% of sub-document requests blocked", rank 3, or "Δ of # sub-documents after blocking", rank 13) and indirectly ("# of `<html>` elements created by blocked scripts", rank 21). Similarly, whether the same `<iframe>` element was used to load multiple sub-documents further predicted page breakage. For example, "Δ in # of sub-documents after blocking" (rank 13) was predictive of breakage, independent of the "# of `<iframe>` on the page" (rank 19).

Third, network behaviors were highly predictive of breakage. Though only five of the 40 predictive features in our set directly related to network requests made by the page, this group contains both the first and second most predictive features ("Δ in bytes sent over network after blocking" and "size of resources directly blocked", respectively). The number and percent of network requests blocked were also, unsurprisingly, predictive of breakage (rank 24 and 32, respectively).

5.2.2 Features without Predictive Power

Additionally, we briefly note some features we expected (i.e. they were "expert curated") to predict page breakage, but which ended up not being predictive. We give below a list of features we expected to be predictive, but which were not.

For example, we expected the number of blocked event registrations (i.e. the number of events that blocked scripts would have registered) to predict page breakage. We expected this on the intuition that at least some of these event registrations would have been important page behaviors (e.g. form handlers, interactive page elements), behaviors that would "break" the page if they were omitted. However, this proved not to be the case; the number of blocked event registrations did not predict breakage. We suspect this might be because most event registrations in blocked scripts end up not being core to page behavior, and are instead more likely to be related to user tracking or other undesirable (to the user) behaviors (e.g. interaction "heat maps" and other fine-grained behavioral tracking).

Similarly, despite our expectations, the number of text nodes inserted by blocked scripts did not predict page breakage. Our intuition was that if blocking a script removes a lot of text from the page, that script is likely important to the page. However, this turned out to not be the case; this feature was not predictive of breakage. One possibility is that unwanted scripts (e.g. large advertisements, captions for video ads, etc.) end up also being responsible for a large amount of text, and so

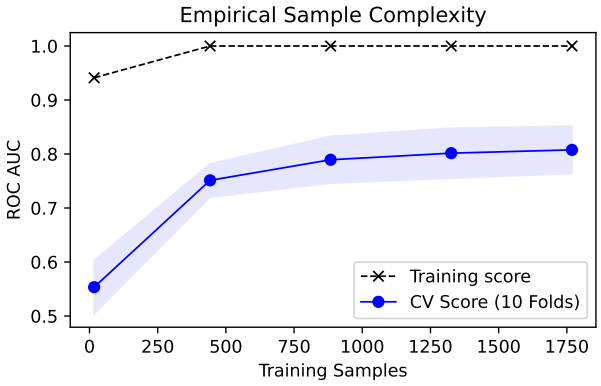

Fig. 7. Empirical sample complexity estimated by the mean ROC-AUC as a function of the number of training samples across 10 folds.

"amount of text added to the page" ends up not being useful for distinguishing blocking that breaks a page from blocking that leaves the page functioning well.

5.3 Sample Complexity

Last, we empirically analyze the sample complexity of the classifier in order to understand the amount of training data needed to achieve practical utility.

Analogous to the previous section we omit the tuning step and initialise the classifier with default parameters. We then train the classifier on varying amounts of data (1%, 25%, 50%, 75% and 100% of training samples) including all features, each time reporting the mean ROC-AUC over 10 folds to estimate classification performance given the respective amount of data. Finally, we plot the mean ROC-AUC as a function of the number of training samples in Figure 7.

We can see that performance converges after training the classifier on roughly 50% of the training data (983 samples), i.e. collecting and training on additional training samples yields diminishing returns. Given the cost of acquiring new samples, this is a desirable result.

6 Discussion

We next provide some discussion of how our system could be deployed, extended and improved by future work. We first discuss practical deployment scenarios for our compatibility classifier, and how our system could be used by filter list authors to improve the compatibility and coverage of crowd sourced filter lists. Next, we discuss how our approach could be extended to other, non-filter list privacy interventions, and the potential difficulties in doing so. We then examine the potential effects of the dynamism of Web content on our measurements, and describe the strategies we employed to mitigate this. Last, we discuss some of the limitations and weaknesses in our approach, and how they could be addressed by future work.

6.1 Deployment Strategies

We here briefly discuss several ways our classifier could be used to improve the usability of content blocking filter lists, and by site authors to prevent the chance their sites break for users of content blocking tools.

6.1.1 Aid Existing Filter List Auditors

One possible deployment strategy for a web-compat classifier is to use the classifier as an aid to existing human evaluators. In this case, the classifier could be tuned to prefer recall over precision, and could be used to reduce the number of broken sites a human evaluator would need to consider; the human labeler would still need to distinguish broken cases from working cases, but the number of cases the human labeler would need to consider would be reduced. The classifier would be, effectively, a force-multiplier for existing filter list developers.

More concretely, the classifier could be used to crawl the Web and look for potential breaking websites. Filter list developers and maintainers would still be needed to distinguish the false positives from true positives (optimizing for recall would reduce precision), but the number of cases humans evaluators would need to consider would be significantly reduced, from "the universe of all possible websites" to "the false positives generated by the classifier."

6.1.2 Protecting Must-Work Sites

Similarly, the classifier could be deployed to warn when a new rule might break "high-priority" sites, as part a continuous-integration-style system for filter list development. Different communities of filter list users (or developers) may have sites that are of extra-importance to them (e.g. high popularity sites, either globally or by linguistic community, or sites vital to communication or

safety with threatened groups, etc). In such cases, filter list authors might wish to be extremely confident that these priority sites continue to work when new filter list rules are added. Manually checking such sites every time a new filter list rule is added would be prohibitive (popular filter lists are updated several times a day). An automated classifier could reduce the amount of manual verification needed to manageable levels.

6.1.3 Assisting Site Authors

Finally, the classifier could also be used by site authors who wish to be warned when a new filter list rule might break their site for users of content blocking tools. While we expect that most site authors would prefer visitors not use content blocking tools at all, the popularity of "please disable your content blocker" notifications (often with a "dismiss" option for users) suggest that a non-trivial number of site authors would prefer their sites work in the presence of a content blocking tool, over their site breaking for the user all together. In these cases, concerned site authors could use the classifier to receive an "early warning" when their pages might break for filter list users.

Site hosting services, or reverse-proxy services (like Cloudflare or Fastly) could offer such "breakage" warnings as a service to their clients.

6.2 Applicability to Other Privacy Interventions

Though we choose to build our classifier to predict when filter list rules could break a page, we expect our approach could be extended to other privacy interventions. For example, Web privacy tools i) modify how third-party storage is managed, ii) use list or heuristic-based approaches to protect users against navigational-tracking, and/or iii) deploy a range of defenses against browser fingerprinting, among many other protections. All of these approaches risk breaking websites, either because they fail to distinguish between benign and malicious behavior, or because the malicious behavior they prevent is deeply entwined with desirable page behaviors.

We expect that a classifier that could detect when these other privacy interventions break sites would be beneficial to their developers and users, for many of the same reasons discussed in Section 2. Developers of these tools could use an automated classifier to detect when, and what kinds of, sites break, and use that information to either refine their tools, or create application-exceptions when needed.

While we expect the general approach of constructing and training our classifier would work for other privacy interventions, extending our classifier to other systems would require some non-trivial changes. For example, our work leverages the EasyList commit history to create a "naturally occuring" ground truth dataset; finding a comparable set of pre-labeled data may be difficult for other projects. Similarly, many of the features we selected for predicting page breakage are likely more applicable to filter-list blocking than other privacy interventions (e.g. changes in number of scripts requested, sub-documents loaded, or event handlers registered). Detecting when other kinds of privacy interventions break pages likely will require other (or at least additional) features.

6.3 Dynamic and Changing Page Behavior

The Web is not deterministic. Visiting the same URL multiple times can return different content, sometimes entirely different pages as time passes and Web sites naturally change and evolve. Moreover, the same content can potentially behave very differently from visit to visit, even if the visual appearance of the page looks unchanged. Recordings of multiple visits to a URL are thus not necessarily comparable: the same browser events will not necessarily occur in both instances, let alone in the same order, introducing noise beyond whatever variable was intentionally altered between the visits (e.g. filter-rule changes). We account for the inherent dynamism of the Web with a multi-pronged approach.

First, we employed multiple levels of validation and pruning when collecting examples for our training dataset to ensure that the historical filter-rule changes we examined still meaningfully applied to the current versions of the Web pages our crawler visited. Filter-rule changes from before 2013 were excluded altogether; extracted URLs which resolved to HTTP error codes were similarly cut from crawling. We then evaluated both the pre-intervention and post-intervention filter rules against each page's recorded network activity, and dropped pages which showed no resulting difference in blocking decisions between the two rulesets (§3.2). A final layer of validation ensured that the pre-intervention PageGraph recorded for each page contained elements connected to network resources blocked by the intervention, implying that the intervention would actually

change page behavior (§4.1). This process pruned our dataset down to the 1,966 examples used to train our classifier.

Second, we avoided computing any single classifier feature from recordings of more than one visit to a page. To compute the difference in the number of bytes sent over the network with and without the intervention, we could have totaled up the transferred bytes tracked in each of the pre- and post-intervention PageGraph recordings, then subtracted the one from the other. But because these recordings represent separate page loads, other factors beyond the intervention could change page behavior between visits by our crawler and influence any difference in these numbers. For example, ad and tracking networks make decisions involving complex markets of competing parties and thousands of data-points to choose how to respond to Web requests in realtime [12, 13]. Instead of comparing numbers between the pre- and post-intervention graphs, we computed features like this from the *intervention-only graph* detailed in Section 3.3. Briefly, the intervention-only graph for a page is the subgraph of the pre-intervention PageGraph containing recorded page behavior which can be attributed to resources that would be blocked by the post-intervention filter rule set but are not blocked by the pre-intervention rules. Then the number of bytes transferred as recorded by the intervention-only graph gives us a measure of the network activity impacted by the blocking behavior of the intervention. In this way, we measure the effects of the intervention without allowing avenues for the Web's dynamism to enter into our calculations. These features ranked highly in predictivity in our analysis presented in Table 2.

6.4 Limitations and Future Work

Finally, we note some weaknesses and limitations in our approach, and suggest how they could be improved through future work. First, our crawler does not interact with pages when recording (through PageGraph) page behaviors. As a result, there are likely cases where our crawler does not trigger some "broken" behaviors on the page, causing those broken behaviors to not be recorded in our dataset. This means that our classifier is not considering some (possibly) predictive information, and so is not performing optimally.

For example, consider the case when a filter list rule blocks a form validation script, but that form validation script is only applied to the page after some user interaction (such as clicking on a "contact us" button on the page). Our crawler would record the script being blocked, but *not* how the page behaves when the user tries to submit the now-broken contact form. As a result, certain categories of broken behaviors are missed by our crawler and classifier. Future work in this area might address this weakness (and so likely improve the performance of the classifier) by having the crawler interact with pages, either by having the instrumented browser be driven by a human user, or through software that attempts to simulate human interactions[15].

Second, and more broadly, while our system can tell filter list authors when a site might be breaking, our system does not provide the filter list author with an easy way of fixing the site. Filter list authors could choose to remove the relevant rule, or modify the rule so that the rule is not applied to the breaking site. This would maintain compatibility, but only by undermining the initial goal of the filter list! Figuring out how to modify filter list rules so they protect privacy without sacrificing compatibility is beyond the scope of work, but is an important area for future work.

Third, we note that there are other kinds of features that could be used to possibly further improve classifier performance. Our implementation only considered page behaviors when trying to predict breakage, but our approach could be extended to consider other available data. For example, features could be designed to consider visual differences in a page before and after blocking, on the (possible) intuition that if applying a filter list rule causes a large visual difference, its more likely the filter list rule has broken the page. Future work could try boosting the classifier's performance with many such other sources of information.

Finally, because we screened out filter-rule changes which had no observable effect on network content-blocking decisions (§3.2), our study ignores changes which touch only so-called "cosmetic" filter rules. These rules inject small snippets of CSS styling into pages to touch up their appearance, predominantly to hide loaded elements and fix up blank spaces left by blocked page components like banner ads. We feel comfortable omitting such rules because, in our experience, they break pages far less often than network rules. The impact of blocking a network resource can cascade outward, preventing scripts from running or causing errors in loaded scripts, leading in turn to failures in page construction and interactive functionality; style changes, on

15 For example, stress-testing scripts like https://github.com/marmelab/gremlins.js.

the other hand, don't have the same knock-on effect on page behavior. Moreover, our focus is on improving Web privacy, and style rules do not (generally) provide users with privacy protections: instead of stopping unwanted content from loading or preventing privacy-invading code from executing, they simply cover up the visible effects.

7 Related Work

In this section we discuss how our automated compatibility classifier compares and relates to other work in the area, and specifically to existing research exploring how the compatibility risks of privacy interventions, and the challenges and history of maintaining content blocking filter lists.

7.1 Compatibility of Privacy Protections

Our work most directly relates to a focused but important area of research and practice around the compatibility costs of privacy enhancing techniques.

Much of the existing work in this area starts with proposed method of improving privacy for users, and then evaluating the compatibility costs of the proposed intervention. For example, Yu et al.[14] proposed an automated system for detecting trackers on the Web, based on how often third-parties reoccurred across first-party sites. They then built an extension that would block detected trackers, and then estimated how many websites their extension broke based on how often users reloaded pages. Snyder et al.[15] similarly suggested that Web privacy and security could be improved by removing infrequently used Web APIs, and had human labelers evaluate how many websites broke when each feature was blocked. Smith et al. proposed improving Web privacy with an automated system that would rewrite scripts to remove privacy harming behaviors without disrupting benign, user serving code paths. They too evaluated their system with human labelers. Iqbal et al. 2020[16] and 2021[17] used similar human evaluation systems for determining the compatibility impact of machine learning based approaches for blocking tracking scripts and detecting fingerprinting scripts, respectively.

Other research has focused on evaluating the compatibility trade-offs in existing Web systems, instead of evaluating the compatibility of a newly proposed system. Jueckstock et al.[18] used a human labeling system to evaluate how often different systems for managing third-party storage in Web browsers broke sites. Mesbah et al.[19], Choudhary et al.[20] and Van Deursen et al.[21] proposed systems for measuring when differences in browser implementations of Web standards broke websites.

Finally, recent work by Mandalari et al.[22] describes a system that determines when privacy interventions break user desirable systems, but for internet-of-things devices instead of websites. Their system automatically distinguishes necessary traffic flows from non-core flows, and only applies privacy protections (i.e. blocking) to traffic flows not necessary for a devices user-serving functionalities.

7.2 Filter List Maintenance

Our work also builds on a large body of work identifying and/or addressing difficulties in maintaining content blocking filter lists. Snyder et al.[23] measured how much "dead weight" (i.e. non-useful rules) had accumulated in popular filter lists, and proposed a system for optimizing filter lists by removing rules that were not ever applied during automated crawls of the Web. Chen et al.[24] proposed a system for detecting when trackers evade filter lists by moving, combining, or renaming tracking scripts by identifying scripts by their behaviors (instead of their URLs). They proposed using their approach to automatically add "evading" scripts to existing filter lists. Sjösten et al.[25] found that many region-specific filter lists were not as well maintained as filter lists targeting languages with more global speakers (e.g. English, Spanish, Chinese, etc), and proposed a machine-learning approach for augmenting regional filter lists based on what is blocked by global lists. Bhagavatula et al.[26] proposed a system for assisting filter list authors by using machine learning to detect textual patterns in blocked URLs, and to use that classifier to generate new filter list rules.

Alrizah et al.[27] measured how often, and how long it took for blocked scripts to try and evade being blocked by filter lists, and found that it often takes filter list authors over a month to respond to evasion efforts. Wang et al.[28] similarly found filter lists have difficulty keeping up when websites attempt to evade detection, though their work focused on websites modifying page structure to avoid cosmetic filtering rules.

Finally, a body of work has studied the difficulties filter list authors face when sites attempt to block filter list users (e.g. "anti-ad-block", or "ad-block-blockers").

Iqbal et al.[29] and Nithyanand et al.[30], for example, both find that many sites attempt to detect when a visitor is applying a filter list (either by checking for blocked requests or for hidden page elements) and apply a range of countermeasures to try and coerce the visitor to disable their content blocking tool.

8 Conclusion

In this work we have presented the first accurate and fully automated system for classifying whether applying a filter list rule to a website would break the user-desirable features on that website. Past work has documented the significant privacy, performance, and security benefits of filter-list-based blocking, but such work only counts the "benefits" side of the ledger. Absent a way of systematically predicting the "costs" of adding more privacy protections, privacy research risks becoming detached from reality. Without a scalable way of estimating compatibility risk, more blocking, more filtering, and more interventions will always look better. If the usability costs are ignored, a broken system will always appear more private than a functioning one.

We hope our work is a useful step towards finding practical, scalable ways of detecting when privacy interventions break the systems they aim to improve. Our work focuses on filter lists rules (because filter lists are among the most popular and well-studied privacy interventions on the Web), but all proposed privacy interventions would benefit from similar systems.

Acknowledgments

This work was partially funded by the NSF under Grant Number CCF-1918573, by Brave Software, and by a gift from Intel.

References

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

[2] A. Gervais, A. Filios, V. Lenders, and S. Capkun, "Quantifying web adblocker privacy," in *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2017, pp. 21–42.

[3] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang, "Knowing your enemy: understanding and detecting malicious web advertising," in *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2012, pp. 674–686.

[4] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna, "The dark alleys of madison avenue: Understanding malicious advertisements," in *Proceedings of the Conference on Internet Measurement Conference (IMC)*, 2014, pp. 373–380.

[5] K. Garimella, O. Kostakis, and M. Mathioudakis, "Ad-blocking: A study on performance, privacy and counter-measures," in *Proceedings of the ACM on Web Science Conference*, 2017, pp. 259–262.

[6] E. Pujol, O. Hohlfeld, and A. Feldmann, "Annoyed users: Ads and ad-block usage in the wild," in *Proceedings of the Internet Measurement Conference (IMC)*, 2015, pp. 93–106.

[7] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2016, pp. 785–794.

[8] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction*. Springer, 2009, vol. 2.

[9] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[10] J. Lei, M. G'Sell, A. Rinaldo, R. J. Tibshirani, and L. Wasserman, "Distribution-free predictive inference for regression," *Journal of the American Statistical Association*, vol. 113, no. 523, pp. 1094–1111, 2018.

[11] M. Smith, P. Snyder, B. Livshits, and D. Stefan, "Sugarcoat: Programmatically generating privacy-preserving, web-compatible resource replacements for content blocking," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[12] C. Castelluccia, L. Olejnik, and T. Minh-Dung, "Selling off privacy at auction," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[13] Y. Yuan, F. Wang, J. Li, and R. Qin, "A survey on real time bidding advertising," in *Proceedings of 2014 IEEE International Conference on Service Operations and Logistics, and Informatics*. IEEE, 2014, pp. 418–423.

[14] Z. Yu, S. Macbeth, K. Modi, and J. M. Pujol, "Tracking the trackers," in *The Web Conference (WWW)*, 2016.

[15] P. Snyder, C. Taylor, and C. Kanich, "Most websites don't need to vibrate: A cost-benefit approach to improving browser security," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[16] 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)*. IEEE, 2020, pp. 763–776.

[17] U. Iqbal, S. Englehardt, and Z. Shafiq, "Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[18] J. Jueckstock, P. Snyder, S. Sarker, A. Kapravelos, and B. Livshits, "Measuring the privacy vs. compatibility trade-off in preventing third-party stateful tracking," in *The Web*

Conference (WWW), 2022.

[19] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[20] S. R. Choudhary, "Detecting cross-browser issues in web applications," in *33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011.

[21] A. Van Deursen, A. Mesbah, and A. Nederlof, "Crawl-based analysis of web applications: Prospects and challenges," *Science of Computer Programming*, vol. 97, pp. 173–180, 2015.

[22] A. M. Mandalari, D. J. Dubois, R. Kolcun, M. T. Paracha, H. Haddadi, and D. Choffnes, "Blocking without breaking: Identification and mitigation of non-essential iot traffic," *Proceedings on Privacy Enhancing Technologies (PETS)*, vol. 4, pp. 369–388, 2021.

[23] P. Snyder, A. Vastel, and B. Livshits, "Who filters the filters: Understanding the growth, usefulness and efficiency of crowdsourced ad blocking," *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS*, 2020.

[24] 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)*. IEEE, 2021, pp. 1715–1729.

[25] A. Sjösten, P. Snyder, A. Pastor, P. Papadopoulos, and B. Livshits, "Filter list generation for underserved regions," in *Proceedings of The Web Conference (WWW)*, 2020, pp. 1682–1692.

[26] 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*, 2014, pp. 95–102.

[27] M. Alrizah, S. Zhu, X. Xing, and G. Wang, "Errors, misunderstandings, and attacks: Analyzing the crowdsourcing process of ad-blocking systems," in *Proceedings of the Internet Measurement Conference (IMC)*, 2019, pp. 230–244.

[28] W. Wang, Y. Zheng, X. Xing, Y. Kwon, X. Zhang, and P. Eugster, "Webranz: web page randomization for better advertisement delivery and web-bot prevention," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 205–216.

[29] U. Iqbal, Z. Shafiq, and Z. Qian, "The ad wars: retrospective measurement and analysis of anti-adblock filter lists," in *Proceedings of the Internet Measurement Conference (IMC)*, 2017, pp. 171–183.

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

A Appendix

Parameter	Default Value	Tuning Range
base_score	0.5	(0.45, 0.55)
colsample_bylevel	1	(0.8, 1)
colsample_bynode	1	(0.8, 1)
colsample_bytree	1	(0.8, 1)
gamma	0	(0, 5)
learning_rate	0.3	(0.2, 0.4)
max_delta_step	0	(0, 5)
max_depth	6	(4, 8)
min_child_weight	1	(1, 5)
n_estimators	100	(80, 120)
num_parallel_tree	1	(1, 5)
reg_alpha	0	(0, 5)
reg_lambda	0	(1, 5)
scale_pos_weight	1	(0.8, 1)
subsample	1	(0.8, 1)

Table 3. XGBoost hyper-parameters with tuning ranges. Ranges are chosen so as to explore configurations in the neighbourhood of parameter default values. These parameters are described in more detail in the XGBoost documentation[16].

16 https://xgboost.readthedocs.io/en/stable/python/python_api.html