SugarCoat: Programmatically Generating Privacy Preserving, Web-Compatible Resource-Replacements for Content Blocking Tools

Anonymous Author(s)

ABSTRACT

Content blocking systems today exempt thousands of privacy-harming scripts. They do this because blocking these scripts breaks the websites that rely on them. In this paper, we eliminate this privacy/functionality trade-off with SugarCoat, a tool that allows filter list authors to automatically patch JavaScript scripts to restrict their access to sensitive data according to a custom privacy policy. We designed SugarCoat to generate *resource replacements* compatible with existing content blocking tools, including uBlock Origin and the Brave Browser, and evaluate our implementation by automatically replacing scripts exempted by the 6,000+ exception rules in the popular EasyList, EasyPrivacy, and uBlock Origin filter lists. Crawling a sample of pages from the Alexa 10k, we find that SugarCoat preserves the functionality of existing pages—our replacements result in Web-compatibility properties similar to exempting scripts—while providing privacy properties most similar to blocking those scripts. Our source code and generated resources are open source, and scripts produced by SugarCoat are being integrated into Brave's content blocking tools.

CCS CONCEPTS

• **Security and privacy → Browser security**; **Usability in security and privacy**; • **Information systems** → *Online advertising*.

KEYWORDS

web privacy, web browsers, web compatibility, content blocking, privacy and usability

ACM Reference Format:
Anonymous Author(s). 2021. SugarCoat: Programmatically Generating Privacy Preserving, Web-Compatible Resource-Replacements for Content Blocking Tools. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A growing—and already large—fraction of Web users (37%) rely on content blockers to prevent unwanted scripts from accessing and tracking private user data [23]. Content blocking extensions like uBlock Origin are some of the most downloaded browser extensions (e.g., they top the charts for both Chrome and Firefox). And browsers like Firefox, Brave, and Edge have even started shipping content blockers built-in and enabled by default.

Thought content blocking significantly improves privacy [12], existing approaches are a far cry from perfect. Most content blocking tools are extremely crude: they make the binary decision to either block or allow a resource according to *filter lists* like EasyList [10]. Unfortunately, the reality of the Web ecosystem doesn't match this binary: some resources are both privacy-harming *and* necessary for page functionality. Filter list authors cannot currently express a more permissive policy like "load resource U, but prevent it from accessing storage", or a more fine-grained policy like "only load the first JavaScript script from resource V, which bundles (concatenates) multiple scripts". This directly impacts the end user: blocking necessary but potentially privacy-invading scripts breaks pages, while allowing them potentially harms privacy.

In response, some content blocking tools—notably, uBlock Origin and the Brave Browser—have added support for *resource replacements*. Instead of simply blocking (or allowing) resources, these tools can be configured to load alternative *safe* resources instead of the original, privacy-harming versions. For example, instead of loading Google Analytics (GA), both uBlock Origin and the Brave Browser load a script that exposes an API that is similar to GA's—ensuring that pages that rely on GA continue to work—but is otherwise inert, and thus does not harm user privacy[1].

While resource replacements can be used to implement policies beyond the crude allow-or-deny binary, this flexibility comes with a serious trade-off: *scalability*. Implementing effective resource replacements requires domain expertise and is largely manual today. For example, the aforementioned GA replacement script was hand-crafted to "mock" the API exposed by Google Analytics and is updated every time Google updates their interface. Resource replacements that depend on implementation details (e.g., a GA script replacement that cannot access privacy-sensitive data like cookies, but retains the script's functionality for tracking the number of visitors) must be updated whenever the original resources are updated. In practice, this means that few scripts are actually replaced—often those easiest to mock. Together, uBlock Origin and the Brave Browser only replace 27 scripts[2]. Meanwhile, the popular filter lists published by EasyList, EasyPrivacy, and uBlock Origin include more than 6,000 exception rules to unblock compatibility-critical scripts. Tens of thousands of scripts remain unaltered and unblocked even though they pose a risk to privacy.

To bridge this gap we developed SugarCoat. SugarCoat allows filter list authors to *automatically* generate privacy-preserving replacements for arbitrary JavaScript scripts. The key insight to eliminating the need for manual analysis and implementation of resource

Conference'17, July 2017, Washington, DC, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]https://github.com/gorhill/uBlock/blob/master/src/web_accessible_resources/google-analytics_ga.js
[2]https://github.com/gorhill/uBlock/blob/master/src/web_accessible_resources

replacements is to focus on and intercept accesses to the *sources* of privacy-sensitive data (e.g., `document.cookie` and `localStorage`). To this end, SugarCoat instruments JavaScript resources (and the resources they create) to restrict their access to sensitive data sources according to a custom policy (e.g., "load script U, but prevent it from accessing storage").

SugarCoat generates resource replacements in two steps. First, we use dynamic analysis to identify code points where JavaScript code uses Web APIs (e.g., functions, constructors, objects, and object properties) that expose sensitive data source (§3).[3] Then, we *repair* the code at these code points to use "mock" implementations of the same APIs, which expose the same interfaces but enforce privacy policies specified by filter list authors.

While this approach shares some similarities with previous work on tracking information flow in the browser [9, 17] and fine-grained policy enforcement for JavaScript [26], it also differs in an important way: we designed SugarCoat to be *backwards-compatible*, *cross-platform*, and *deployable*. As such, SugarCoat generates resource replacements that can be used by any content-blocking tool that supports resource replacements today, including uBlock Origin, the Brave Browser, and AdGuard.

This paper advances the state of content blocking on the Web via four contributions:

(1) The design of SugarCoat, a system for automatically rewriting arbitrary JavaScript scripts to enforce privacy-protecting policies (§3). SugarCoat allows filter list authors to replace JavaScript scripts that cannot be blocked (e.g., because they are necessary for functionality) but are potentially harmful for user privacy with safe alternatives (e.g., scripts that preserve functionality but cannot access sensitive data).
(2) An open-source and easy-to-use implementation of SugarCoat. Our implementation extends PageGraph [35], a browser instrumentation system used to analyze the relationship between the HTTP, DOM and JavaScript layers of Web applications. Specifically, we modify the underlying V8 JavaScript engine to de-alias and deobfuscate JavaScript code to identify source code points where sensitive APIs are used. Our changes have already been integrated into PageGraph proper.
(3) An evaluation of SugarCoat on 198 unique in-the-wild tracking-and-advertising JavaScript code scripts (§4). These scripts are labeled by filter lists as privacy harming (or advertising related), but nevertheless allowed by current connect-blocking tools because blocking them would break Web pages. We find that SugarCoat can rewrite all these scripts, to block access to sensitive data, without significantly impacting functionality or page performance.
(4) The complete dataset of all 198 SugarCoat resource replacements (both before and after rewriting) and associated filter list rules, so that our rewritten scripts can be deployed in existing content blocking tools that support resource replacements.

[3]Though we would ideally do this statically, statically analyzing JavaScript code to identify such code points is notoriously hard—both because JavaScript is highly dynamic and because real-world JavaScript is "obfuscated" via minimization and bundling tools. We discuss this in more detail in Section 3.2.1.

```
// Served from tracker.com/track.js
window.track = (callback) => {
  // Generate tracking token if one
  // doesn't exist.
  if (!localStorage["id"]) {
    localStorage["id"] = Math.random();
  }
  const id = localStorage["id"];
  // Record the page load
  fetch("//tracking.com/rec?id=" + id)
    .then(_ => {
      // If a callback was provided,
      // call it
      if (callback) callback();
    });
};
```

Listing 1: Motivating example of a tracking script which sometimes causes compatibility breakage when blocked.

```
<!-- Page served from example.org -->
<script src="//tracker.com/track.js"></script>
<script>
  setup(); // defined elsewhere.
  track(); // defined in track.js.
</script>
<p>Page start</p>
```

Listing 2: Simple example of a site including a tracking script.

2 MOTIVATION AND BACKGROUND

This section gives a simplified example of how content blocking typically works on the Web, how content blocking can unintentionally break websites, and the limited, unsatisfactory options content blocking tools currently turn to in such cases. The section concludes by outlining the properties needed for a better solution to Web-compatible content blocking.

2.1 Motivating Example

In this section we explain how content blocking works, in the simplest case, and why content blocking sometimes unintentionally breaks websites. These examples are greatly simplified from real world code.

The examples are intended to demonstrate how content blocking tools are often faced with the lose-lose choice of either protecting privacy but breaking a site, or keeping the site working but allowing the privacy harm.

2.1.1 Typical Tracking Script Integration. We begin with a toy tracking script, presented in Listing 1. In this example, the script is served from https://tracker.com/track.js, and defines a global function, `track`. This function generates a unique identifier, persists it in storage, and sends the identifier to a recording service. The `track` function also takes an optional callback function that, if provided, will be called after the tracking has occurred.

Next, Listing 2 presents an example of how a page might integrate this tracking script. In this example, served from https://example.org, the page loads the tracking script. The page then includes an inline script that sets up the page's functionality with a call

```html
<!-- Page served from example.org -->
<script src="//tracker.com/track.js"></script>
<script>
  track(setup);
</script>
<p>Page start</p>
```

Listing 3: Example of content blocking breaking a page.

to a `setup()` function (defined elsewhere), and then performs the privacy-harming operation by calling `track()`.

2.1.2 Typical Content Blocking Scenario. Content blocking tools are well-equipped to protect privacy given the above integration pattern. Once a privacy-harming script has been identified, its URL is added to a common list (e.g., EasyList [10], EasyPrivacy [11]), either verbatim or generalized using regular-expression like patterns. Content blocking tools like uBlock Origin [14] or AdGuard [2] then pull from these centralized lists, so that many content blocker users will benefit from the privacy improvement.

In the above case, a filter list contributor might add the rule `||tracker.com/track.js`. Content blocking tools using this list would then no longer fetch the tracking script, improving privacy and performance for these users.

When person using a content blocking tool re-visits https://example.org (again depicted in Listing 2), now with the new filter rule enabled, the page will execute differently. The content blocking tool will block the request to https://tracker.com/track.js. The `setup()` function will then be called (which, in this example is not affected by blocking). However, because the tracking script was blocked, the `track()` function will not have been defined, and that call will fail. Instead of performing the privacy-harming behavior, an exception will be thrown. Since the page's functionality has already been set up with the `setup()` function, the page will otherwise behave as normal from the perspective of the user.

2.1.3 Typical Breaking Scenario. Finally, we present an alternative example, wherein content blocking causes a page to break. Consider the example in Listing 3, consisting of the same functionality but structured differently. Now, instead of calling `setup()` and then `track()`, `setup` is passed as a callback function to `track`; `setup()` will only run after the `track()` completes successfully.

Without content blocking, the page will run correctly (though privacy will be harmed). https://tracker.com/track.js will be fetched, meaning `track()` will be defined, and so `setup()` will eventually be called. In the content blocking case though, the tracking script is not loaded, so an exception will be thrown; `setup()` will never be called, resulting in a broken page.

2.2 Current Options for Content Blocking

The previous subsection gives an example of how content blocking tools often face the lose-lose scenario of sacrificing either functionality or privacy. In this subsection we describe the options currently available to content blocking tools, and why they are unsatisfactory.

2.2.1 Exception Rules. The simplest, and most common, option is to add an exception, and not block privacy-harming scripts on sites that break. Exceptions have the benefit of requiring the least time and expertise from filter list authors, important because filter lists are often crowdsourced, which limits how much expertise can be required for participation.

However, compatibility-through-exceptions has tremendous downsides. First, exception rules re-enable the privacy harm the tool intended to fix. And second, exception rules creates negative incentives for site authors, by "rewarding" sites that intentionally break in the presence of content blocking.

2.2.2 Manually Developed Resource Replacements. A second approach to is to manually develop alternative implementations of privacy-harming scripts: implementations that remove the privacy-harming functionality but otherwise maintain the code's API "shape". Content blocking tools can load these alternative implementations *in place of* the original code, protecting privacy but otherwise allowing the page to function as normal.

While powerful, resource replacements are not currently a practical, general solution. Generating resource replacements requires significant expertise and time, requiring the understand and reverse engineering of large, minified JavaScript libraries. High development and maintenance costs make manually-developed resource replacements practical for only the most common privacy-harming scripts on the Web.

2.2.3 Sandboxing through Runtime Modifications. A third approach is to modify the JavaScript engine to apply privacy protections at runtime. Scripts labeled as privacy-harming would still be fetched, but "tainted" and given different privileges than other scripts. This approach has the upside of potentially addressing a large number of compatibility concerns, but with prohibitive costs.

First, the JavaScript engine modifications needed to robustly enforce such policies are complex and costly. Labels need to be tracked and propagated to "downstream" code units; scripts should only be as trusted as the scripts that included them. Additionally, a robust solution would need to prevent scripts from loosing their trust labels by "laundering" code through DOM sinks and network requests, and would complicate optimizations that opportunistically defer code compilation, among other concerns. In short, the cost of such a system has (so far) proven prohibitive.

Second, content blocking tools benefit from crowdsourcing, or the contributions of large number of semi-expert contributors. Approaches that only work in one browser would forfeit many of the benefits that crowdsourcing provides, by fracturing the set of possible contributors. Unless all browser vendors implement the kind of runtime taint information needed for a "sandboxing" approach to work (something which seems unlikely for the immediate future), compatibility solutions that require engine modifications will be limited in their breath and usefulness.

2.3 Properties of an Ideal Solution

We outline the properties required for a broad, Web-scale solution to fixing web-compatibility issues in content blocking tools.

First, a robust solution to privacy-vs.-compatibility problems in content blocking tools must **maintain the privacy benefits of current blocking tools**. This implies that privacy-harming code should not be able to access privacy-affecting APIs.

Second, a solution should **minimize impact on benign site behavior**, both in privacy-affecting code (i.e., scripts labeled by filter lists), and in surrounding code. Significantly, this rules out approaches that make global changes to Web APIs that benign page functionality requires. Changes and interventions should be local to privacy-harming scripts.

Third, a solution must be **scalable and automated**, so that it can be applied to the wide range of privacy-harming scripts on the Web. Solutions that require significant expertise and time, such as manually generated resource replacements, will only be able to address a small number of cases.

Finally, a solution should be **broadly compatible with existing browsers**, to maintain the benefit of crowdsourcing filter list generation. Solutions that only work in one browser or one tool will reduce the number of people who can contribute to, test, and maintain the filter lists that many privacy-protecting, content blocking tools depend on.

3 SUGARCOAT DESIGN

The previous section described how content blocking tools often face a lose-lose choice of privacy vs. compatibility. In this section we present the design of SugarCoat, a multi-stage system that programmatically generates privacy-preserving resource replacements using deep browser instrumentation to analyze and alter the behavior of JavaScript code. We first give a high-level overview of the SugarCoat pipeline, and then proceed to describe each step in the process.

3.1 High-level Overview

SugarCoat takes as input a set of target JavaScript code units (i.e., script URLs) considered privacy-harming, and one or more Web pages which include those code units. SugarCoat returns a modified version of the same JavaScript code, rewritten to enforce privacy properties.

Conceptually, SugarCoat does this by:

(1) dynamically observing the execution of target JavaScript code in an instrumented browser, and concretizing privacy-relevant Web API accesses to textual locations in the JavaScript source code;
(2) using static analysis to map each Web API call site to its immediate containing scope and the corresponding JavaScript AST nodes;
(3) rewriting the code at these locations to redirect the privacy-relevant API accesses to "mock" implementations, which have the same API signatures, but apply a privacy-preserving policy; and
(4) generating content-blocker compatible resource replacements from the rewritten code, along with corresponding filter rules to instruct content blockers to load the replacements instead of the original code.

3.2 SugarCoat Pipeline

We now explain each step in the SugarCoat pipeline in detail. The pipeline takes as input a set of *target scripts*—code units for which privacy-preserving replacements should be generated—and the URL of a Web pages that includes the script(s). SugarCoat then generates resource replacements versions of the target scripts through the following process.

Table 1: Privacy-Relevant APIs Targeted by SugarCoat

Network APIs	Description
`fetch`	Modern HTTP request API
`XMLHttpRequest`	Legacy HTTP request API
Storage APIs	**Description**
`document.cookie`	Script access to origin cookies
`localStorage`	Persistent key-value storage
`sessionStorage`	Session-duration key-value storage
`Storage`	Storage interface base class

We target a limited set of privacy-relevant APIs for this work, but note that SugarCoat is designed to be extended to cover arbitrary Web APIs.

3.2.1 Attributing and Concretizing API Calls. SugarCoat first loads and runs the given Web pages in an instrumented browser, to observe the behavior of the target scripts in situ. The browser can either be driven through scripted automation (e.g., Puppeteer [13]), or by a human user. While the page is running, SugarCoat notes i) when a target script accesses a privacy-relevant Web API, ii) the concretized locations of those accesses in the JavaScript source text, and iii) any additional scripts a target script brings into the page (e.g., through `eval`, inserting `<script>` tags, etc.).

The goal of this step is to identify all the locations in each target script where privacy-relevant Web APIs are called. Though conceptually simple, the highly dynamic nature of JavaScript makes statically enumerating these call sites in JavaScript code difficult and fraught with limitations. Obfuscation, minification, and other label-stripping practices common in Web build systems introduce further complications.

We sidestep these difficulties by observing actual script behavior dynamically, instead of trying to predict it statically. We make use of PageGraph [35], a browser instrumentation system for Blink- and V8-based browser engines, which records the page "actions" that occur during page execution (e.g., DOM node modifications, Web API calls, HTTP requests), the "actors" responsible (e.g., the parser, a script unit), and "action receivers" (e.g., DOM nodes, network resources, other actors), along with relevant attributes and metadata. This history of actions is represented as a single, interconnected directed graph.

For example, a script inserting a DOM node is recorded by PageGraph as i) a node representing the script unit (the "actor"), ii) a node representing inserted DOM node (the "action receiver"), and iii) a directed edge connecting the two nodes, representing the insertion (the "action"). Edges and nodes are annotated with additional information, like the source URL and V8 script ID for the script node, and references to parent and sibling nodes for the insertion edge.

In this work, we extend PageGraph to also record concretized source text locations for relevant Web API accesses during a browsing session. Further, we record this data for every script on the stack when an access occurs, so indirect Web API accesses via shared libraries can be traced back to the scripts calling into those libraries. We also expand the number of Web APIs tracked by PageGraph,

```javascript
function getTrackingId (persistent) {
  const storage =
    window[persistent ? "localStorage"
          : "sessionStorage"];
  let trackingId =
    storage.getItem("trackingId");
  if (!trackingId) {
    trackingId = Math.random();
    storage.setItem("trackingId",
      trackingId);
  }
  return trackingId;
}
```

Listing 4: Sample privacy-harming code snippet. The `getTrackingId` function returns a unique tracking identifier for the user, stored to disk (via the `localStorage` API) if the caller requests persistence, or session-only otherwise (via `sessionStorage`).

and make it possible to mark additional APIs for instrumentation by annotating the corresponding interface definition code (i.e., WebIDL). Table 1 lists the Web APIs SugarCoat tracked in this work, though SugarCoat can be used to consider most other Web APIs.

From PageGraph recordings of browsing sessions, SugarCoat extracts the full list of privacy-relevant API accesses that occurred (along with stack data) for each access. SugarCoat filters this list down to accesses either involving target scripts, or a script included by a target script (and scripts that *those* scripts included, recursively). Finally, for each of interest API access, SugarCoat identifies the stack frame belonging to a target or "downstream" script which has most recently been pushed to the stack at the time of the API call, and notes the script unit and concretized source text location for that stack frame.

This stage of the SugarCoat pipeline yields a list of "Web API, script unit, concretized location" tuples, forming the input to the next stage of the pipeline.

3.2.2 *Mapping Concretized Calls to AST Scopes.*

SugarCoat uses the behavioral data to drive a static pass through the target and downstream scripts—hereon referred to collectively as "target scripts".

First, script source code is parsed into abstract syntax trees (ASTs) using ESPrima[4], a high-quality JavaScript parser.

SugarCoat walks each parsed AST, starting from the top-level script scope and descending through nested function scopes. For each scope level, SugarCoat filters the list of "Web API, script unit, concretized location" tuples from the previous pipeline stage, keeping only those for which the current scope is the narrowest, most deeply-nested scope that still contains the concretized location. Whenever this filtered list is non-empty, SugarCoat collects the AST node corresponding to the current scope and the set of Web APIs in that list of accesses.

This stage of the pipeline results in a list of "AST node, set of Web APIs accessed" tuples. For example, for Listing 4, this list could contain the tuple "`function getTrackingId() {...}`, {`localStorage`, `sessionStorage`}". The list of tuples is then provided to the next pipeline stage.

[4]https://esprima.org

```javascript
function getTrackingId (persistent) {
  const storage =
    $mockLocalStorage;
  let trackingId =
    storage.getItem("trackingId");
  if (!trackingId) {
    trackingId = Math.random();
    storage.setItem("trackingId",
      trackingId);
  }
  return trackingId;
}
```

Listing 5: The previous listing, subject to a naive rewriting strategy gone wrong. This hypothetical implementation observed only the `localStorage` pathway while collecting script behavioral data, and so assumed that the value assigned to `storage` (highlighted) always evaluates to `localStorage`. By replacing this expression directly, the meaning of the code has been changed.

3.2.3 *Rewriting JavaScript Units.*

SugarCoat combines the results of dynamic and static analysis of the target scripts to drive the generation of new JavaScript text, implementing privacy-preserving version of the target script(s). Depending on which privacy-relevant APIs a target script is determined to use, a series of manually-written "mock" implementations are tacked onto the beginning of the script, which emulate the expected behavior in a compatible but privacy-preserving way—for example, a Web Storage API mock which keeps all data in memory, or a Fetch API mock which returns fake responses for network requests. Given the AST nodes and associated lists of APIs from the previous stage, this stage of the pipeline is tasked with rewriting each AST node so that calls to the specified APIs are replaced with calls to the injected mock versions.

A naive approach to performing this rewriting would be to perform an in-place replacement of the exact JavaScript expressions encoding Web API accesses (e.g., `window.localStorage`) with expressions that access the mocks (e.g., `$mockLocalStorage`). However, this approach can result in a fragile system: Listing 5 demonstrates how it could unintentionally change the meaning of the code from Listing 4, potentially breaking compatibility. Furthermore, the sheer dynamism of the JavaScript language produces an explosion of edge cases to handle, making *correctly* implementing the most obvious rewriting strategy prohibitively difficult.

Our approach is to wrap each target scope with *entry* and *exit guards*, so that, while control flow is in the target scope, references in the JavaScript environment to the target Web APIs are temporarily replaced with their mock equivalents. This is demonstrated in Listing 6. The original code is wrapped in a `try/finally` block; when control flow enters the block, *entry guards* overwrite `window.localStorage` and `window.sessionStorage` with mocks; when control flow exits the block, *exit guards* restore what they originally pointed to. Since the previous pipeline stage selected the only narrowest scopes containing each recorded Web API access, the spill-over effects of these temporary replacements are limited.

An added benefit of our scope-based approach is that accesses to privacy-relevant APIs can be redirected even when the accesses

```
function getTrackingId (persistent) {
  try {
    $replace(window, "localStorage",
      $mockLocalStorage);
    $replace(window, "sessionStorage",
      $mockSessionStorage);
    const storage =
      window[persistent ? "localStorage"
        : "sessionStorage"];
    let trackingId =
      storage.getItem("trackingId");
    if (!trackingId) {
      trackingId = Math.random();
      storage.setItem("trackingId",
        trackingId);
    }
    return trackingId;
  } finally {
    $restore(window, "localStorage");
    $restore(window, "sessionStorage");
  }
}
```

Listing 6: A high-level illustration of SugarCoat's rewriting strategy, applied to the tracking code from Listing 4. Injected code is highlighted. The `localStorage` and `sessionStorage` APIs are temporarily overwritten with mock implementations, so the function body code—even though it has not been changed—will access those mocks when it runs.

themselves are performed by separate, shared libraries like jQuery—shared libraries which may be used legitimately by other, non-privacy-harming scripts on the page, and therefore aren't targeted for rewriting. As discussed in Section 3.2.2, when a target script calls into a shared library, and that shared library calls a privacy-affecting API on behalf of the target script, we do not modify library code; instead, we inject mocks in the calling-target-script *before* control is transferred to the library, and then remove those mocks *after* the library returns control to the target script.

We note that the generated code presented in Listing 6 is simplified for the purposes of explanation. We take additional steps to ensure SugarCoat preserves the semantics of the code's original context. For example, injected variable names are uniquely randomized, variable declarations are hoisted outside the enclosing wrapper blocks where necessary, function-default-parameters are specially handled, and function body code is wrapped in immediately invoked function expressions (IIFEs) to preserve JavaScript lexical semantics.

3.2.4 *Generating Resource Replacements.*

As a final step, SugarCoat turns the rewritten AST back into JavaScript source code, packages up the rewritten scripts into a resource replacement bundle, and generates accompanying EasyList-style filter rules[5] to intercept requests to the original scripts and redirect them to the rewritten versions.

The output can be dropped into any compatible content blocking tool, such as uBlock Origin [14], AdGuard [2], or the Brave Browser's `adblock-rust`[6] engine.

[5]https://github.com/gorhill/uBlock/wiki/Static-filter-syntax#redirect
[6]https://github.com/brave/adblock-rust

Table 2: Summary of Data Gathered to Evaluate SugarCoat

Crawl Configuration	
Measurement period	11/25 – 12/04/2020
Filter lists used	EasyList, EasyPrivacy, uBlock Origin, Brave
# exception rules	6,405
# pages recorded	902
JavaScript Crawl Statistics	
# script units loaded	20,981
… matching exception rules	3,034
… rewritten by SugarCoat	1,701
# API calls intercepted by SC	139,589
… storage API calls	130,494
… network API calls	9,095

4 EVALUATION

This section presents an evaluation of SugarCoat across multiple dimensions. We first describe the dataset used throughout the evaluation. We then present two measurements of SugarCoat's impact on privacy, two measurements of SugarCoat's web-compatibility properties, and conclude with a performance evaluation.

4.1 Terminology

The following terms are used throughout this section.

Privacy-relevant APIs: JavaScript APIs that can be used to harm user privacy. For the purposes of this work, we consider only the APIs in listed in Table 1, though we note that SugarCoat could be easily extended to wrap most other Web APIs.

Blocked script: A JavaScript resource blocked because of a filter list rule.

Excepted script: A JavaScript resource that would have been blocked because of a filter list rule, but which is instead allowed to load because of an additional exception rule. Excepted scripts are typically scripts that would break a page if blocked.

Rewritten script: A script that has been rewritten with the SugarCoat pipeline, with the intent of preventing its from accessing privacy-relevant APIs, but otherwise not changing the script's functionality.

"Default", "blocked", and "rewritten" pages: Pages where target scripts are respectively loaded (because of an exception rule), blocked (and so prevented from being fetched or executed), or rewritten (using the SugarCoat versions of scripts in place of the original scripts). We note that in all three cases, all scripts other than target scripts are blocked (or not) according to the original filter list rules.

4.2 Evaluation Dataset

The privacy, compatibility, and performance measurements presented in this section all draw from the same dataset, consisting of 902 popular Web pages measured under three conditions. All measurements were performed from a residential IP address in California, using an instrumented, Chromium-based browser driven by the Puppeteer[7] automation library. For each page crawled, we waited for the document's `onload` event to trigger, and then waited a further 15 seconds, to allow scripts to execute.

4.2.1 *Web Page Selection.*

We started by collecting the URLs of Web pages on which privacy-harming scripts would have been blocked by filter lists, were it not for compatibility concerns. We generated this dataset in several steps.

First, we crawled each site in the Alexa 10k and recorded all network activity occurring on the landing page. We then randomly selected a same-site link on the page (i.e., a link pointing to another page on the same eTLD+1), repeating the process a maximum of four times, yielding a maximum of five page URLs per site, including the landing page.

Second, we reduced this collection of URLs to only pages that included at least one excepted script. We did so by using the `adblock-rs` library[8] to apply the most popular filter lists (noted in Table 2) to the network requests captured from each page. We then excluded all pages that did not include at least one excepted script.

Third, we further reduced the set of pages to those where an excepted script accessed at least one privacy-relevant API. We did so by re-crawling the remaining page URLs in a PageGraph-enabled browser. As described in Section 3.2.1, PageGraph records what scripts access which Web APIs, and which scripts bring additional "downstream" scripts into the page. For each re-crawled page, we looked for any instances of privacy-relevant APIs being accessed either i) directly, by an excepted script or ii) indirectly, by a "downstream" script injected by an excepted script. We randomly sampled 902 such pages, marking the relevant scripts as "target" scripts for rewriting by SugarCoat.

4.2.2 *Measurements of Selected Pages.*

We visited each of the collected Web pages under three conditions, to determine how page execution differs when the privacy-relevant target scripts are excepted, blocked, and rewritten.

Every condition used the same instrumented PageGraph-enabled browser, but differed in the set of filter rules fed into the browser's content blocking engine:

(1) **"Default" rule set:** the full set of rules assembled from popular filter lists, ensuring that the excepted target scripts would be fetched and executed.
(2) **"Blocked" rule set:** the full set of filter list rules, but with the relevant exception rule(s) removed, ensuring that the previously-excepted scripts would now be blocked.
(3) **"Rewritten" rule set:** the full set of filter list rules, but with the relevant exception rule(s) exchanged for resource replacement rules, ensuring that SugarCoat-rewritten versions of the target scripts would be loaded and executed *instead of* the original, privacy-harming versions.

For each visit under each condition, we recorded the full resulting PageGraph output (including all DOM modifications, network requests, script executions, etc), the original target script text, and the resource replacements generated by SugarCoat. Table 2 summarizes the properties of the crawl data.

[7]https://developers.google.com/web/tools/puppeteer/
[8]https://www.npmjs.com/package/adblock-rs

Table 3: Script-Level Privacy Evaluation

	Original	Rewritten
# Storage Calls (Mean)	77	0
# Storage Calls (Median)	17	0
# Storage Calls (Total)	130,494	0
# Network Calls (Mean)	5	0
# Network Calls (Median)	0	0
# Network Calls (Total)	9,095	0

Measurements of the number of storage and network calls made by excepted scripts when executed on the pages that include them, both in their original form, and after being rewritten by SugarCoat.

Finally, we revisited each of the 902 pages in a typical (i.e., non-instrumented, non-PageGraph-enabled) browser, once with the "default" rule set and again with the "rewritten" rule set. We used this additional crawl to measure the performance of pages using SugarCoat-generated resource replacements, as compared to the original excepted scripts, and to demonstrate that SugarCoat generated resource replacements can be used by popular content blocking extensions in "stock", non-modified Web browsers.

4.3 Privacy Evaluation

We conduct two measurements of SugarCoat's effect on privacy using the dataset described in the previous section. We first measure how effectively SugarCoat removes privacy-relevant behaviors from target scripts (and from their downstream dependencies). We then measure the overall privacy impact of using SugarCoat-generated resource replacements on a page, by comparing the total number of privacy-relevant behaviors between "default" and "rewritten" versions of pages.

4.3.1 *Script Level Evaluation.*

First, we measure how effectively SugarCoat improves privacy *at the script level*, comparing the recorded behavior of the original versions of the web-compatibility-critical scripts in our evaluation dataset with the behavior of the rewritten versions generated by SugarCoat. Table 3 presents this comparison, generated by counting the number of storage and network API calls made by each original script versus its rewritten counterpart. JavaScript storage and network APIs are grouped as presented in Table 1. We find that SugarCoat dramatically cuts the number of privacy-affecting behaviors scripts engage in.

4.3.2 *Page Level Evaluation.*

Next, we assess the page-level impact of SugarCoat by measuring the differences in privacy-relevant behaviors between "default" and "rewritten" pages (as defined in Section 4.1). For each of the 902 pages in our dataset, we extract total counts of JavaScript storage and network API calls from the PageGraph data, for both the default and rewritten conditions. Figure 1 presents this measurement as overlapping cumulative distribution functions (CDFs) of these counts.

We find that using SugarCoat-generated resource replacements in place of the original target scripts results in significantly fewer privacy-relevant behaviors on each page, while the use of the same

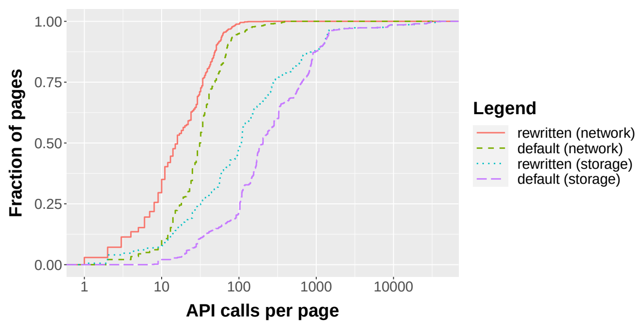

Figure 1: CDFs of the number of storage and network JavaScript APIs called on 902 pages, when privacy-affecting scripts are excepted and run normally ("default"), and when they are replaced with SugarCoat-rewritten versions ("rewritten").

APIs by non-privacy harming (i.e., non-target) scripts is largely unaffected.

We further note that this comparison likely *under-emphasizes* the privacy improvements SugarCoat provides. All network requests and storage operations do not carry the same risks; they can be used for privacy-neutral or privacy-harming purposes. The scripts that SugarCoat targets in these measurements have been identified by filter list authors as particularly privacy threatening (i.e., they were blocked at one point) but then never the less allowed to load, generally for web compatibility purposes. We therefore suspect that the storage and network operations in the target, excepted scripts are particularly likely to be privacy harming, and so using SugarCoat to prevent those operations is particularly likely to be privacy enhancing.

4.4 Web Compatibility Evaluation

We next present two measurements which confirm that, while SugarCoat provides similar privacy benefits to blocking target scripts, it does so *without* harming desirable page functionality.

4.4.1 Qualitative Compatibility Evaluation. We first evaluated SugarCoat's effect on Web compatibility though a qualitative, double-blind, manual evaluation, adopting an approach from previous work[33]. This process also served as a test of applying SugarCoat to address real-world Web compatibility issues introduced by content blocking.

We selected 50 live sites to serve as test cases, pulled from Web compatibility issue logs reported to the EasyList filter list project. [9] Each selected site included at least one script that was both i) originally labeled as harmful by EasyList, and ii) explicitly allowed to load because of an exception rule introduced by EasyList developers to fix the corresponding compatibility issue. We used SugarCoat to generate rewritten, privacy-preserving versions of these web-compatibility-critical scripts, and produced three configurations for each site:

(1) a "default" configuration, with the full, unmodified filter lists;
(2) a "blocking" configuration, removing the critical exception rule from the filter list, and thus blocking the critical scripts; and
(3) a "rewritten" configuration, exchanging the exception rule for resource replacement rules, and loading the rewritten version of each critical script in place of the original.

To assess the compatibility impact of replacing these critical scripts with SugarCoat-rewritten versions, we recruited six human evaluators with no financial or professional relationship to the authors. Each site was tested independently by two of the six evaluators. We instructed each evaluator to interact with the site for one minute each time, under three test conditions. First, the evaluator visited the site in the "default", known-working configuration, to learn what functionality the site provides. Next, the evaluator was presented with either the "blocking" configuration (expected to show compatibility breakage, as privacy-harming but web-compatibility-critical scripts were blocked), or the "rewritten" configuration (expected to behave similarly to the default configuration from a compatibility perspective). Finally, the evaluator was presented with the remaining unseen configuration. The order in which the blocking and rewritten configurations were presented to the evaluator was randomized, with a 50% chance of the evaluator seeing either configuration first; evaluators were never told if they were seeing the blocking or the rewritten configuration.

For each of the blocking and rewritten configurations, the evaluator ranked the site's functionality on a scale of 1 – 3:

(1) There was no perceptible difference between the configuration presented and the default, control configuration.
(2) The browsing experience was altered, but the evaluator was still able to complete the same tasks as during the control visit.
(3) The evaluator was not able to complete the same tasks as during the control visit.

Table 4 presents the results of this evaluation.

Our results support several conclusions. First, filter lists accurately identify privacy-harming but compatibility-critical scripts. Second, it's often unambiguous when a site is broken, given the

Table 4: Results of Qualitative Compatibility Evaluation

Measure		
# evaluators		6
# sites evaluated		50
% agreement		90%
Measure	**Mean**	**Median**
When blocking	2.86	3
With SugarCoat	1.03	1

Comparison of how often 6 independent evaluators considered a site "broken" when blocking a privacy-harming but compatibility-critical script, and when using a SugarCoat rewritten version of the script. "1" meant worked like normal, "3" meant completely broken.

[9]The EasyList project notes fixes to compatibility issues by prefacing those git commit messages with "P:", as can be seen in the project's commit history at https://github.com/easylist/easylist/commits.

Table 5: Results of Quantitative Compatibility Evaluation

Privacy-affecting behaviors	Case I	Case II
Storage	< 0.001	< 0.001
Network	< 0.001	< 0.001
Core functionality behaviors	**Case I**	**Case II**
DOM operations	0.016	0.732
Event registration	0.007	0.517

P-scores from our two-sample K-S tests, comparing the distributions of:

Case I: Pages with privacy-harming but compatibility-critical scripts loaded vs. pages with such scripts blocked.

Case II: Pages with privacy-harming but compatibility-critical scripts loaded vs. pages with such scripts rewritten by SugarCoat.

high level of agreement between our evaluators. And third, using SugarCoat to rewrite critical scripts, instead of blocking them, leads to significantly less Web compatibility breakage.

4.4.2 Quantitative Compatibility Evaluation. We also conducted a quantitative evaluation of SugarCoat's compatibility impact, by comparing the aggregate behavior of "blocked" pages to "rewritten" pages. Combined with our other evaluations, we believe these measurements suggest that SugarCoat provides much of the privacy improvement of blocking target scripts, while significantly reducing the risk of compatibility breakage.

We conducted this evaluation by measuring the distributions of Web API calls under different blocking conditions. These measurements were drawn from the dataset described in Section 4.2. Using the PageGraph data collected from the "default", "blocked", and "rewritten" versions of each page (as defined in Section 4.1), we counted the numbers of calls to different Web APIs, per page, for each measurement condition. We clustered the instrumented APIs into a smaller number of purpose categories, to better capture developer goals distinct from implementation choices. One category, "DOM operations", includes APIs involved in creating, inserting, removing, and updating DOM nodes, representing the actions scripts take to build or modify a page's structure. Another category, "event registrations", groups APIs used to attach and manipulate event listeners (e.g., `addEventListener`, `.on[event]`), representing page interactivity.

We applied the two-sample Kolmogorov-Smirnov (K-S) test to determine the likelihood that differences observed in API call counts under different measurement conditions reflect different underlying distributions, and so different underlying page behavior[10]. Table 5 presents the results of these K-S tests. We find that blocking privacy-harming-but-necessary scripts has a significant ($p < .05$) effect on both privacy-relevant and non-privacy-relevant page behaviors. We also find that applying SugarCoat-generated resource replacements (instead of blocking) maintains the significant ($p < .05$) effect on privacy-relevant page behaviors, but no longer has a significant effect on core functionality page behaviors.

[10]We opted for K-S testing over Student's T-test because our data is not normally distributed, as determined by standard normality testing.

Table 6: Results of Performance Evaluation

Target Script Sizes	
Original Script Size	158,081 bytes
Delta Script Size	+10,102 (+6%)
JavaScript Memory Usage (Entire Page)	
Original JS Heap Used Size	10,842,772 bytes
Delta JS Heap Used Size	+12,928 (+0.001%)
Performance Event Timing (Entire Page)	
Delta Time to DOM Content Loaded	-5 ms
Delta Time to DOM Interactive	-3 ms
Delta Time to Load Event	-22 ms
Delta Time to First Paint	-5 ms
Delta Time to First Contentful Paint	-8 ms

Measurements comparing page performance with privacy-harming scripts rewritten by SugarCoat against the same pages running the original versions of the scripts. All measures are median values. Positive deltas indicate an increase relative to the original versions; negative deltas indicate a decrease or speed-up.

The results of this quantitative evaluation support several conclusions. First, blocking privacy-relevant but compatibility-critical scripts significantly reduces the number of both privacy-relevant (e.g., storage, network) and core-functionality (e.g., document manipulation, event registration) operations on a page. And second, using SugarCoat to rewrite (instead of block) these scripts maintains the statistically significant reduction in privacy-relevant behaviors, but no longer causes a statistically significant reduction in non-privacy-related page behaviors. This supports the finding of the manual evaluation that using SugarCoat to rewrite scripts significantly reduces compatibility issues.

4.5 Performance

Finally, we evaluated the performance overhead of using SugarCoat-generated scripts in place of their original implementations, using the crawl data from Section 4.2. Since SugarCoat injects code to produce rewritten versions of the original scripts, some file size increase is expected; we measured the size of this increase. We also measured the impact of SugarCoat-rewritten scripts on JavaScript memory usage and the timing of key page performance events, using existing APIs provided by Puppeteer.

These measurements are reported in Table 6, which summarizes the differences observed between "rewritten" and "default" versions of each page in the evaluation dataset. We conclude from these measurements that SugarCoat-rewritten scripts are slightly larger, and use slightly more memory than, their original counterparts, but perform equivalently through the lens of page performance timing. We expect that this performance overhead would be negligible for most users, under most conditions.

5 DISCUSSION AND LIMITATIONS

The previous sections present the design of SugarCoat, and how we evaluated the privacy, compatibility and performance characteristics of the system. In this section we discuss some of the limitations of SugarCoat, how SugarCoat might be deployed in practice, and possible future directions for this work.

5.1 Limitations

5.1.1 Compatability Measurements. Because compatibility is a subjective evaluation, its difficult to measure compatibility with the techniques common to privacy research. Section 4.4 provides two very different attempts to measures compatibility. We note here some reasons why each approach only provides a partial answer. The manual evaluations in Section 4.4.1, for example, may have completeness issues; its possible that the replacement scripts introduced compatibility problems that the evaluators would have encountered if they interacted with the sites for longer. We believe developing better techniques for understanding how privacy interventions affect desirable application behaviors (both on the Web and otherwise) is an area deserving of future work.

5.1.2 Ground Truth. The evaluation presented in Section 4 assumes that filter lists accurately distinguish privacy-harming resources from benign resources. The evaluation further assumes that filter lists accurately identify cases where filter rules break websites, and that filter list authors address such situations with exception rules. While we know that these assumptions are not 100% correct, we believe this generalization is still useful, for multiple reasons. First, this assumption is inline with much existing research that treats filter lists as an imperfect-but-best-possible source of ground truth (see [4, 5, 15, 20], among many others). And second, the fact that many-millions of people use these lists suggests, even if only anecdotally, that the lists are at least accurate enough to be useful. Nevertheless, we note this assumption as a limitation, and suggest that establishing high quality ground truth in this area as a possible area for future work.

5.1.3 Rewriting Limitations. While we believe SugarCoat to be an effective way to solve compatibility issues from filter-list based content blocking, we note some limitations. Most significantly, SugarCoat relies on behavioral analysis when concretizing API calls, and so carries with it all limitations of behavioral analysis systems, including only being able to understand code paths that execute during observation. This limitation can be partially addressed by trying to execute many code paths as possible (through human interaction, guided-discovery, etc.), but these techniques only lessen the limitation, they do not fundamentally solve it.

Second, in some scenarios SugarCoat could introduce new compatibility problems. If the version of a library rewritten by SugarCoat, and the version of the library expected by surrounding code, fall out of sync, pages may break if they depend on features added in the interim. While this limitation is important, we believe it can be addressed through regular application of SugarCoat (to keep replacements up to date), or generating per-version replacements (when "versioned URLs" are available).

Third, our approach to rewriting scripts is to identify the code locations that access privacy-affecting APIs (or call into library code that does the same), inject mock, privacy preserving implementations of the relevant API before the call site, and then remove the mock API implementation once control returns to the target script. In practice we find this approach generally keeps the API modifications "local" to the call site.

However, in our current implementation, there are scenarios where side effects could leak into other parts of the application. For example, if a target script calls an "async" library function that uses a privacy-affecting API, any other code the executes between when the target script calls the function and when the promise resolves will also see the mock API implementation, regardless of its location in the application. This decision reflects an intentional preference for coverage over compatibility at the margin; an earlier version of omitted handing async and similar calls into libraries for this reason. SugarCoat users could similarly decide to only handle synchronous calls if it better suited their needs. However, in practice, we did not find any instances where this had a noticeable impact (partially because the async methods in our mock APIs resolve immediately, further reducing the possible timing window). We nevertheless note the tradeoff and limitation for completeness.

Last, applying SugarCoat on a true Web scale would require dramatically more on-device storage than is typically used by content blocking tools. We discuss this limitation more, along with possible solutions, in the next section.

5.2 Deployment Scenarios

Content blocking tools that use resource replacements store the entire catalogue of scripts on the client. This is not currently a significant constraint; resource replacements are still rarely used (on the scale of dozens), for reasons explained in Section 2.2).

SugarCoat's is designed to generate replacements on a Web scale though, which will require alternative deployment strategies on resource constrained devices, devices without sufficient storage to carry a complete catalogue of thousands of alternative scripts. Maximizing SugarCoat benefit on resource-constrained devices requires may require alternative deployment strategies.

Resource constrained devices may choose to still pre-fetch resource replacementss, but to only do so for some threshold of popular sites. Second, clients could instead fetch resource replacements as needed from a central repository, and use a private information retrieval (PIR) scheme to avoid revealing leaking browsing habits. Third, replacement scripts could be distributed not as complete alternative implementations, but as patches to the original scripts. Each of these strategies allow the client to trade coverage, performance and privacy against each other, as best suits the capabilities of the device.

5.3 Other Applications

In this work we've focused on using SugarCoat to generate resource replacements to improve the web compatibility of content blocking tools. We here finally note that these same techniques could be used server side, or by application developers.

For example, with few exceptions, the Web platform does have a way to include as script in a page, but with limited capabilities. A site author, for example, may wish to include some third-party library, but prevent it from calling the network. For many developers, auditing, understanding, and rewriting heavily minified or obfuscated JavaScript is prohibitive. SugarCoat could be used by such authors to easily restrict the capabilities of included scripts (keeping in mind the caveats and limitations discussed in Section 5.1).

6 RELATED WORK

This work contributes to and builds on a large amount of existing work into Web privacy, filter lists (and how advertisers and trackers respond to being included on filter lists), and efforts to constrain, analyze and evaluate Web applications. We in this section we discuss how our work relates to existing research in these areas.

6.1 Filter Lists

This work is closely related to a large field of research into the benefits from, effectiveness of, and reactions to filter list based content blocking. Work such as Merzdovnik et al. [24] found that filter list based content blocking tools often result in a net reduction in CPU use, even when accounting for the overhead of the tools themselves. Gervais et al. [12] found that similar content blockers reduce communication with third-parties by as much as 40% in typical browsing scenarios. Several works have documented shown that tracker and advertisers often try to circumvent or deter filter list use [19, 28, 40, 43], though Iqbal et al. [19] found that filter lists are often effective at defeating these tracker-counter measures (e.g., anti-adblock scripts). Storey et al. [38] prove a model of tracker-vs-blocker interactions and conclude that blockers are likely to prevail in the cat-and-mouse game.

Other work has found that, though useful, filter lists contain significant inefficiencies. Snyder et al. [34] found that most rules in the most popular filter lists provide no benefit. Similarly, Alrizah et al. [3] found that popular filter lists contain non-trivial numbers of false positives.

Finally, filter lists have been used to establish ground truth in the training and evaluation of many other content blocking strategies. Chen et al. [5] used filter lists as ground truth to detect tracking scripts based on behavioral signatures, and Iqbal et al. [20] used lists to train a classifier-based blocking system. Several other works have used filter lists to train systems that block ads based on their visual appearance [1, 30, 38].

6.2 Platform-Wide Web Privacy Improvements

Filter lists improve Web privacy by deploying defenses against known bad-actors. A different line of work aims to improving Web privacy through platform wide changes and interventions.

For example, much work has focused on changing browsers' storage polices to improve Web privacy. Roesner et al. [32] early on documented the benefits of blocking third-party storage all together. More recently, some browsers have moved to a hybrid strategy: blocking some forms of third-party storage completely (i.e., cookies), and isolating other kinds of third-party storage with "partitioning" strategies. For example, Apple's Safari browser [11], limits cross-site tracking by partitioning third-party storage per first-party, per browser-session. The Tor Browser Bundle and recent versions of Firefox implement similar partitioning strategies [12]. Jueckstock et al. [21] recently variant strategy, of partitioning third-party storage per first-party, per site-session, to further limit certain forms of cross-session tracking.

A related vein of work focus on platform-wide (instead of actor-specific) defenses against browser fingerprinting. Nikiforakis et al. [27] proposed a system for preventing fingerprinting based on randomizing the values of some Web APIs. Laperdrix et al. [22] improved the Privaricator proposal by using stenographic-like techniques to minimize the impact of randomization on user-benefiting functionality. Olejnik et al. [29] found that feature removal can be an effective privacy-preserving technique, for features that are rarely used and have a high potential for abuse.

6.3 Web Compatibility

Our work also draws from, and relates to, a large body of work investigating the compatibility impact of privacy tools. Mesbah et al. [25] and Choudhary [7] both proposed system for detecting whether a page works correctly when executing in different browser engines. Deursen et al. [39] design a system for detecting broken applications by looking differences in traversable links in pages.

Other works have used manual evaluations of sampled websites to determine whether a page is working correctly. Snyder et al. [33] use repeated, double blind manual evaluations of websites to determine whether a given Web API was necessary for a page to function correctly. Iqbal et al. [18] used manual evaluations of webpages to determine whether anti-fingerprinting protections broke pages, Jueckstock et al. [21] used a similar technique for measuring the impact of different browser storage polices on web compatibility. Yu et al. [42] used a novel, indirect method of detecting compatibility issues, by measuring how often users disabled their privacy-focused tool.

6.4 Language-based Confinement Techniques

Dynamic information flow control systems like FlowFox [9] and JSFlow [17] can be used to prevent privacy-sensitive data flow leaking. Both FlowFox and JSFlow cannot easily be deployed, one requires multiple runs of the browser, the other imposes roughly 100% overhead. Other dynamic enforcement systems for JavaScript (e.g., ConScript [26]) require heavy modification to browsers. On the opposite side of the spectrum, static information flow and static analysis doesn't work well either: JavaScript is too dynamic.

Still, our work is similar in spirit to Chugh et al.'s [8], which performs static analysis to make dynamic information flow control practical, but we instead use dynamic analysis to inform a static pass that instruments code with guards. Our instrumentation is similar to work on program repair (e.g., [31]), program partitioning (e.g., SIF [6]), and policy weaving (e.g., [16]).

Systems like BFlow [41] and COWL [37] help developers write applications that do not leak sensitive data, but require significant modifications to applications. Unfortunately, we can't expect ad-block developers to do this.

[11]https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/

[12]https://wiki.mozilla.org/Security/FirstPartyIsolation

7 CONCLUSION

Content blocking has become an important part of a privacy-, performance- and user-respecting Web, so much so that some government security agencies recommend content blocking tools to government employees, to prevent some forms of attack [36]. Unfortunately, websites increasing put content blocking tools in no-win situations: continue protecting user-privacy but render an increasingly large number of sites as "broken", or maintain compatibility but by allowing the privacy harm the user intended to avoid by installing the content blocking tool.

In this work we present SugarCoat, an automated, practical system to shift the state of the Web back in favor of content blocking tools (and so, back in favor of users). SugarCoat gives provides a solution to these (previously) no-win situations; maintain privacy and compatibility through the programmatic generation of privacy-preserving resource replacements. SugarCoat is intended for real-world use, and so is designed to be compatible with existing popular content blocking tools. Further, as part of this work, we share the full implementation of SugarCoat [13], the full set of 198 resource replacements [14] generated by applying SugarCoat to all exceptedscripts encountered on our sample of 902 popular websites, and the corresponding filter list rules [15] needed to use these resource replacements in existing content blocking tools.

Finally, we emphasize that SugarCoat is designed to be a real-world, deployable, practical system, designed to increase the amount of control users have over their use of the Web. SugarCoat is currently being integrated by at least one privacy-focused browser vendor, and we are actively working with the maintainers of popular content blocking tools so that SugarCoat can benefit their users too. We hope that SugarCoat will be a useful tool to keep users in control of their Web browsing, and contribute to future work empowering Web users.

REFERENCES

[1] Zainul Abi Din, Panagiotis Tigas, Samuel T King, and Benjamin Livshits. 2020. {PERCIVAL}: Making in-browser perceptual ad blocking practical with deep learning. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 387–400.

[2] AdGuard. [n.d.]. AdGuard. https://adguard.com/.

[3] Mshabab Alrizah, Sencun Zhu, Xinyu Xing, and Gang Wang. 2019. Errors, Misunderstandings, and Attacks: Analyzing the Crowdsourcing Process of Ad-blocking Systems. (2019).

[4] Sruti Bhagavatula, Christopher Dunn, Chris Kanich, Minaxi Gupta, and Brian Ziebart. 2014. Leveraging Machine Learning to Improve Unwanted Resource Filtering. In *ACM Workshop on Artificial Intelligence and Security*.

[5] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. [n.d.]. Detecting Filter List Evasion With Event-Loop-Turn Granularity JavaScript Signatures. In *Proceedings of the IEEE Symposium on Security and Privacy (May 2021)*.

[6] Stephen Chong, Krishnaprasad Vikram, Andrew C Myers, et al. 2007. SIF: Enforcing Confidentiality and Integrity in Web Applications.. In *USENIX Security Symposium*. 1–16.

[7] Shauvik Roy Choudhary. 2011. Detecting cross-browser issues in web applications. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 1146–1148.

[8] Ravi Chugh, Jeffrey A Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged information flow for JavaScript. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 50–62.

[9] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 748–759.

[10] Famlam Fanboy, MonztA and Khrin. [n.d.]. EasyList. https://easylist.to/easylist/easylist.txt.

[11] Famlam Fanboy, MonztA and Khrin. [n.d.]. EasyPrivacy. https://easylist.to/easylist/easyprivacy.txt.

[12] Arthur Gervais, Alexandros Filios, Vincent Lenders, and Srdjan Capkun. 2017. Quantifying web adblocker privacy. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10493 LNCS (2017), 21–42. https://doi.org/10.1007/978-3-319-66399-9_2

[13] Google. [n.d.]. Tools for Web Developers: Puppeteer. https://developers.google.com/web/tools/puppeteer/.

[14] gorhill. [n.d.]. uBlock Origin. https://github.com/gorhill/uBlock.

[15] David Gugelmann, Markus Happe, Bernhard Ager, and Vincent Lenders. 2015. An Automated Approach for Complementing Ad Blockers' Blacklists. In *Privacy Enhancing Technologies Symposium (PETS)*.

[16] William R Harris, Somesh Jha, and Thomas Reps. 2012. Secure programming via visibly pushdown safety games. In *International Conference on Computer Aided Verification*. Springer, 581–598.

[17] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 1663–1671.

[18] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. 2020. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. *arXiv preprint arXiv:2008.04480* (2020).

[19] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. 2017. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. *ACM SIGCOMM Conference on Internet Measurement Conference (IMC)* 13 (2017). https://doi.org/10.1145/3131365.3131387

[20] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. 2020. Adgraph: A graph-based approach to ad and tracker blocking. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 763–776.

[21] Jordan Jueckstock, Peter Snyder, Shaown Sarker, Alexandros Kapravelos, and Benjamin Livshits. 2020. There's No Trick, Its Just a Simple Trick: A Web-Compat and Privacy Improving Approach to Third-party Web Storage. *arXiv preprint arXiv:2011.01267* (2020).

[22] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. 2017. FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *International Symposium on Engineering Secure Software and Systems*. Springer, 97–114.

[23] M Malloy, M McNamara, A Cahn, and P Barford. 2016. Ad blockers: Global prevalence and impact. *Imc'16* 14-16-Nove (2016), 119–125. https://doi.org/10.1145/2987443.2987460

[24] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar Weippl. 2017. Block Me if You Can: A Large-Scale Study of Tracker-Blocking Tools. *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017* (2017), 319–333. https://doi.org/10.1109/EuroSP.2017.26

[25] Ali Mesbah and Mukul R Prasad. 2011. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*. 561–570.

[26] Leo A Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 481–496.

[27] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. 2015. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web*. 820–830.

[28] Rishab Nithyanand, Sheharbano Khattak, Mobin Javed, Narseo Vallina-Rodriguez, Marjan Falahrastegar, Julia E. Powles, Emiliano De Cristofaro, Hamed Haddadi, and Steven J. Murdoch. 2016. Ad-Blocking and Counter Blocking: A Slice of the Arms Race. *CoRR* abs/1605.05077 (2016). arXiv:1605.05077 http://arxiv.org/abs/1605.05077

[29] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. 2015. The leaking battery. In *Data Privacy Management, and Security Assurance*. Springer, 254–263.

[30] Adblock Plus. [n.d.]. Sentinel - the artificial intelligence ad detector. https://adblock.ai/.

[31] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid information flow control. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–30.

[32] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. 2012. Detecting and defending against third-party tracking on the web. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 155–168.

[33] Peter Snyder, Cynthia Taylor, and Chris Kanich. 2017. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 179–194.

[34] Peter Snyder, Antoine Vastel, and Ben Livshits. 2020. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking.

[13]https://github.com/SugarCoatWeb/sugarcoat
[14]https://github.com/SugarCoatWeb/sugarcoat-data/tree/master/resources
[15]https://github.com/SugarCoatWeb/sugarcoat-data/blob/master/rules.txt

Proceedings of the ACM on Measurement and Analysis of Computing Systems 4, 2 (2020), 1–24.

[35] Brave Software. [n.d.]. PageGraph. https://github.com/brave/brave-browser/wiki/PageGraph.

[36] Tim Starks. [n.d.]. CISA tells agencies to consider ad blockers to fend off 'malvertising'. https://www.cyberscoop.com/ad-blockers-security-nsa-dhs-wyden/.

[37] Deian Stefan, Edward Z Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. 2014. Protecting Users by Confining JavaScript with {COWL}. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 131–146.

[38] Grant Storey, Dillon Reisman, Jonathan Mayer, and Arvind Narayanan. 2017. The future of ad blocking: An analytical framework and new techniques. *arXiv preprint arXiv:1705.08568* (2017).

[39] Arie Van Deursen, Ali Mesbah, and Alex Nederlof. 2015. Crawl-based analysis of web applications: Prospects and challenges. *Science of computer programming* 97 (2015), 173–180.

[40] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. 2016. WebRanz: web page randomization for better advertisement delivery and web-bot prevention. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016* (2016), 205–216. https://doi.org/10.1145/2950290.2950352

[41] Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. 2009. Privacy-preserving browser-side scripting with BFlow. In *Proceedings of the 4th ACM European conference on Computer systems*. 233–246.

[42] Zhonghao Yu, Sam Macbeth, Konark Modi, and Josep M Pujol. 2016. Tracking the trackers. In *Proceedings of the 25th International Conference on World Wide Web*. 121–132.

[43] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, and Heng Yin. 2018. Measuring and disrupting anti-adblockers using differential execution analysis. In *The Network and Distributed System Security Symposium (NDSS)*.