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

Michael Smith
UC San Diego, USA

Pete Snyder
Brave Software, USA

Benjamin Livshits
Brave Software, USA

Deian Stefan
UC San Diego, USA

ABSTRACT

Content blocking systems today exempt thousands of privacy-harming scripts. They do this because blocking these scripts breaks the Web sites that rely on them. In this paper, we address 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. SugarCoat is intended for real-world practical deployment, to protect Web users from privacy harms current tools are unable to protect against. Our design choices emphasize compatibility with existing tools, policy flexibility, and extensibility. SugarCoat is open source and is being integrated into Brave's content blocking tools: an initial set of SugarCoat-generated resource replacements are already shipping to users in the Brave Browser.

CCS CONCEPTS

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

KEYWORDS

Web privacy; Web compatibility; Content blocking

ACM Reference Format:
Michael Smith, Pete Snyder, Benjamin Livshits, and Deian Stefan. 2021. SugarCoat: Programmatically Generating Privacy-Preserving, Web-Compatible Resource Replacements for Content Blocking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460120.3484578

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 [25]. 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.

Though content blocking significantly improves privacy [12], existing approaches are far 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 does not 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 privacy-invading scripts breaks pages, while allowing them 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 in place 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 by hand 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

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

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 replacements is to focus on and intercept accesses to the *sources* of privacy-sensitive data (e.g., Web APIs like `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 sources.[3] Then, we *repair* the code at these code points to instead 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, 18] and fine-grained policy enforcement for JavaScript [28], 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, AdGuard, and the Brave Browser, which is already shipping an initial set of SugarCoat-generated resource replacements with its built-in content blocker.

This paper makes four contributions to content blocking:

- 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 potentially privacy-harming JavaScript scripts that cannot be blocked (e.g., because they are necessary for functionality) with safe alternatives: equivalent scripts that preserve functionality but cannot access sensitive data.
- An open-source [4] and easy-to-use implementation of SugarCoat. Our implementation extends PageGraph [37], 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 upstream.
- An evaluation of SugarCoat on 231 unique in-the-wild tracking-and-advertising JavaScript scripts (§4). These scripts are labeled by filter lists as privacy harming (or advertising related), but nevertheless allowed by current content blocking tools because blocking them would break Web pages. SugarCoat can rewrite all these scripts, blocking access to sensitive data, without significantly impacting functionality or page performance.
- The complete dataset [5] of SugarCoat resource replacements and associated filter list rules generated for this work, so that our rewritten scripts can be deployed in existing tools.

[2]https://github.com/gorhill/uBlock/blob/master/src/web_accessible_resources
[3]Though we would ideally do this statically, statically analyzing JavaScript 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.
[4]https://github.com/SugarCoatJS/sugarcoat

```
// Script served from https://tracker.com/track.js
window.track = callback => {
  // Generate tracking token if one does not exist.
  if (!localStorage["id"]) {
    localStorage["id"] = Math.random();
  }
  // Record the page load.
  fetch("//tracker.com/record?id=" + localStorage["id"])
    .then(_ => {
      // If a callback was provided, call it.
      if (callback) callback();
    });
};
```

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

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

Figure 2: Simple example of a Web page 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 Web sites, 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 and why content blocking sometimes unintentionally breaks Web sites.

2.1.1 Typical Tracking Script Integration. We begin with a toy tracking script, presented in Figure 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, Figure 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 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] or EasyPrivacy [11]), either verbatim or generalized using regular-expression-like patterns. Content blocking tools like uBlock Origin [19] and AdGuard [2]

[5]https://github.com/SugarCoatJS/sugarcoat-paper-dataset

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

Figure 3: Example of content blocking breaking a page.

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 prevent the tracking script from being fetched and executed, improving privacy and performance for these users.

When a person using a content blocking tool visits https://example.org (again depicted in Figure 2) 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 error 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 Figure 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 `setup` will eventually be called. In the content blocking case, however, the tracking script is not loaded, so an error will be thrown at the call to `track`; `setup` will never be called, resulting in a broken page.

2.2 Current Options for Content Blocking

The previous section gives an example of how content blocking tools are forced to sacrifice either functionality or privacy. Next, we describe the different ways content blocking tools can try to address this problem, and the limitations of each approach.

2.2.1 *Exception Rules.*

The simplest and most common option is to add additional filter rules which carve out *exceptions* for scripts that would otherwise be blocked, allowing the privacy-harming scripts to load on Web sites that break without them. Exceptions have the benefit of requiring the least time and expertise from filter list authors. This is important because filter lists are often crowdsourced, which limits how much expertise can be required for participation.

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

2.2.2 *Manually Developed Resource Replacements.*

A second approach is to manually develop alternate 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.

Unfortunately, generating resource replacements is a manual, time-consuming task that requires significant domain expertise: privacy engineers need to understand and reverse-engineer large, minified JavaScript libraries. This, coupled with high maintenance costs to keep up when the original scripts change, makes manually developing 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 is prohibitively expensive.

First, the JavaScript engine modifications needed to robustly enforce such policies are complex and costly. Labels need to be tracked and propagated to "downstream" scripts: scripts should only be as trusted as the scripts that included them. Additionally, a robust solution would need to prevent scripts from loosing their 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 breadth and usefulness.

2.3 Properties of an Ideal Solution

We outline the properties of a general, 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 allowed to access privacy-affecting APIs.

Second, a solution should **minimize impact on benign Web site behavior**, both within privacy-affecting code (i.e., scripts labeled by filter lists), and within surrounding code. This rules out approaches that make global changes to Web APIs; 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 (e.g., manual resource replacement development) can only address a few cases.

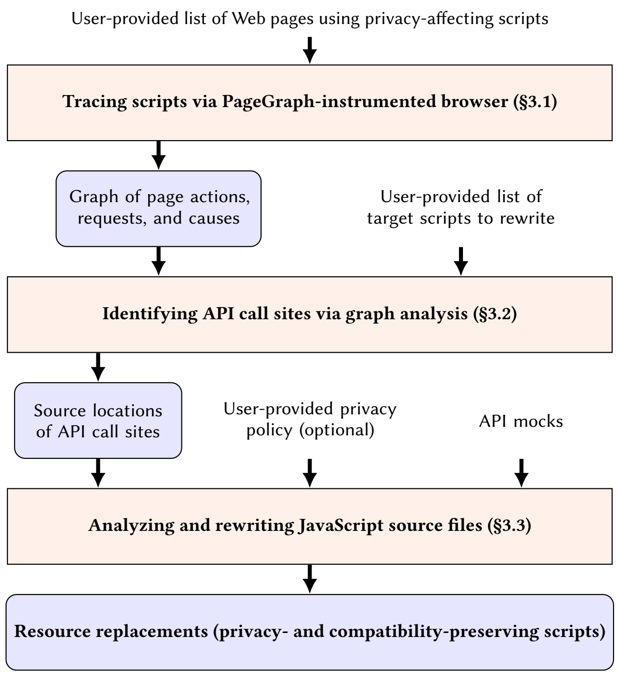
User-provided list of Web pages using privacy-affecting scripts
Tracing scripts via PageGraph-instrumented browser (§3.1)
Graph of page actions, requests, and causes
User-provided list of target scripts to rewrite
Identifying API call sites via graph analysis (§3.2)
Source locations of API call sites
User-provided privacy policy (optional)
API mocks
Analyzing and rewriting JavaScript source files (§3.3)
Resource replacements (privacy- and compatibility-preserving scripts)

Figure 4: Generating privacy-preserving resource replacements with SugarCoat. The user records the execution of Web pages using a PageGraph-instrumented browser. The produced graphs and user-supplied list of target privacy-harming scripts are then used to identify the locations of calls sites within those scripts where privacy-sensitive Web APIs are accessed. Finally, these locations, together with a set of privacy-preserving API mocks (provided by SugarCoat, but customizable) and optional user-supplied privacy policies, are used to generate rewritten scripts. These privacy- and compatibility-preserving rewritten scripts can then be used as resource replacements with existing content-blocking tools like uBlock Origin and the Brave Browser.

Finally, a solution should be **backwards compatible with existing browsers**, to maintain the benefits 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 content blocking tools rely on.

3 SUGARCOAT DESIGN

In this section we present the design of SugarCoat, a system for programmatically generating privacy-preserving resource replacements. SugarCoat combines dynamic browser instrumentation with static code analysis to patch out the privacy-harming portions of real-world JavaScript code. Privacy developers can use SugarCoat to solve the privacy/compatibility trade-off without manually reverse-engineering scripts or writing individual resource replacements.

Generating resource replacements with SugarCoat is a three-step process (see Figure 4):

- The privacy developer visits Web pages using our modified PageGraph-instrumented browser, which dynamically traces the execution of all scripts embedded by the visited pages (§3.1).
- The developer marks certain scripts that they consider privacy-harming, and feeds this *target script set* into the first stage of the SugarCoat pipeline. Using graph analysis, this stage builds behavioral profiles of the target scripts from the collected PageGraph browser data, concretizing privacy-relevant Web API accesses to textual locations in the JavaScript source (§3.2).

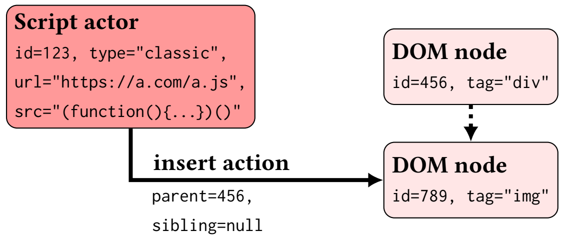

Figure 5: PageGraph representation of a script inserting a DOM node at runtime.

- The second pipeline stage produces resource replacements by analyzing and rewriting the source code of the target scripts to redirect the identified API accesses to "mock" implementations, which have the same API signatures, but apply a privacy-preserving policy (§3.3).

All of these steps are performed "offline" by privacy developers. The generated resource replacements are then deployed to end users in existing browser content blocking tools.

3.1 Tracing Scripts with PageGraph

SugarCoat uses recordings of how privacy-harming scripts behave at runtime to drive the generation of non-privacy-harming resource replacements. Privacy developers collect this behavioral data by visiting Web pages in a modified browser equipped with PageGraph [37], an instrumentation system for Blink- and V8-based browser engines. The browser can either be driven manually or by scripted automation (e.g., in our evaluation we use Puppeteer [13]).

For all pages loaded in the instrumented browser, PageGraph records page "actions" that occur during execution (e.g., DOM node modifications, Web API calls, HTTP requests), the "actors" responsible for the actions (e.g., the parser, running scripts), and the "receivers" which are acted upon (e.g., DOM nodes, network resources, other actors), along with relevant attributes and metadata. This history of actions is represented as an interconnected directed graph, with nodes representing actors and receivers, and edges representing actions as well as the DOM tree relationships in the page. Figure 5 illustrates the graph structure resulting from a script inserting a DOM node at runtime, recorded by PageGraph as (a) a node representing the script (the "actor"), (b) a node representing the inserted DOM node (the "receiver"), (c) an edge connecting the two nodes, representing the insertion (the "action"), and (d) an edge connecting the inserted DOM node to its new parent DOM node. The nodes and edges are annotated with metadata, like the source URL and V8 script ID for the script actor node, and references to parent and sibling nodes for the insertion action edge.

For this work, we extended PageGraph with additional capabilities for tracking the Web API accesses performed by scripts. By hooking into the JavaScript binding layer, PageGraph can now track accesses to arbitrary Web APIs as actions in the graph, as long as simple annotations are added to the WebIDL code defining the APIs (see Appendix C for an example). For each access, we record the concretized JavaScript source text location within every script on the stack at the point the access occurs, saving this as metadata in the graph. This data is collected for all scripts embedded in pages visited with the PageGraph browser, and then extracted from the PageGraph recordings by the SugarCoat pipeline.

```
// tracking.js
function initializeTracking () {
  function getTrackingId (persistent) {
    const storage =
      window[(persistent ? "local" : "session") + "Storage"];
    let trackingId = storage.getItem("trackingId");
    if (!trackingId) {
      trackingId = Math.random();
      storage.setItem("trackingId", trackingId);
    }
    return trackingId;
  }
  ...
  return { getTrackingId, ... };
}
```

Figure 6: Simple privacy-harming script. 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`).

3.2 Identifying Call Sites via Graph Analysis

The first stage of the SugarCoat pipeline builds behavioral profiles of privacy-harming scripts. It takes as input a set of *target scripts*, in source code form, and PageGraph recordings of pages which embed the target scripts. SugarCoat matches the target script source code with script actor nodes in the PageGraph graphs. It then performs two graph traversals: one to *expand the target script set*, and one to *generate a trace map* to drive script rewriting later in the pipeline.

3.2.1 Expanding the Target Script Set. A script can dynamically inject other scripts into the page at runtime—this is a common pattern for ad and tracking scripts in particular. Blocking one script has the knock-on effect of blocking the scripts it would have injected into the page if allowed to run. These additional scripts may not be in filter lists, but may still be privacy-harming. To provide equivalent coverage without blocking, SugarCoat expands the input target script set to include all scripts injected by another target script (and all scripts injected by *those* scripts, recursively).

The most common method by which scripts inject other scripts is to insert `<script>` tags into the DOM. SugarCoat looks for this pattern in the input PageGraph graphs, following insertion action edges out from target script actors to find the DOM nodes that they insert at runtime, then following "execute" action edges out from the inserted DOM nodes to reveal scripts that they cause to be executed. These scripts can then be added to the target script set.

3.2.2 Trace Map Generation. SugarCoat generates non-privacy-harming resource replacements from the target scripts by rewriting their source code to "neutralize" calls to privacy-relevant APIs. To drive the rewriting stage, SugarCoat needs to identify the textual locations of these calls in the source code of the target scripts. The highly dynamic nature of JavaScript makes it hard to statically enumerate these call sites. Obfuscation, minification, and other label-stripping practices common in Web build systems make this harder. We sidestep these difficulties by observing actual script behavior dynamically, instead of trying to predict it statically.

Consider the simple privacy-harming script `tracking.js` in Figure 6. The function `getTrackingId` in this script uses either the `localStorage` or the `sessionStorage` API to store and retrieve a user tracking identifier. It accesses these APIs conditionally and indirectly, without explicitly naming either API in the source text. This makes it hard to statically identify the call site. But at runtime the browser engine knows exactly when these APIs are called and which scripts are on the stack when the calls occur. Our extended version of PageGraph records these calls in the graph data (Figure 7).

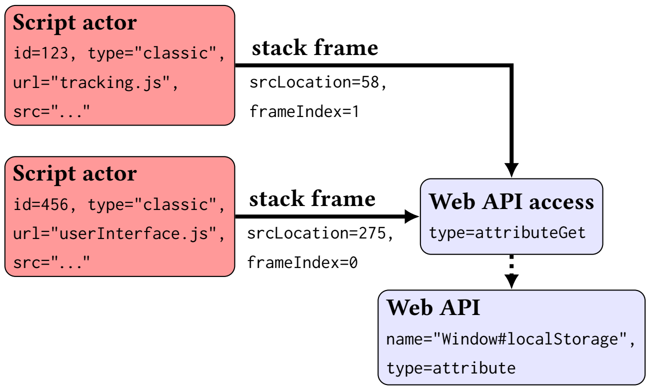

Figure 7: PageGraph representation of a Web API access. The scripts on the stack at the time of the access are linked by edges to the access node, recording their order on the stack and the locations within the JavaScript source text.

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 Web APIs but design SugarCoat to be extensible to cover arbitrary APIs.

Then instead of guessing which pieces of a script might correspond to privacy-relevant API calls, SugarCoat mines the input PageGraph recordings for calls observed at runtime. For each "Web API access" graph node linked to a privacy-relevant API, SugarCoat loops through the connected scripts, starting with the script most recently pushed to the JavaScript stack at the time of the call. The call is attributed to the first script that is in the target script set. In Figure 7, for example, `tracking.js` is at the top of the stack when a call is logged to `localStorage`, and if `tracking.js` is in the target script set, then SugarCoat attributes this call to `tracking.js` at source location 58 (i.e., the 58th character in the script source text). Otherwise, `userInterface.js` is next on the stack, and is checked for its membership in the target script set; and so on.

The output of this process is a trace map linking target script source locations to sets of privacy-relevant APIs accessed at those locations, in the form $(\textit{target script}, \textit{code location}) \longrightarrow \{\textit{Web API}, \ldots\}$.

3.3 Analyzing and Rewriting JavaScript

Next, SugarCoat transforms the original source code of the target privacy-harming scripts into non-privacy-harming resource replacements, by redirecting privacy-relevant Web API accesses to harmless "mock" implementations of those APIs. Source code locations where these accesses occur are drawn from the trace map produced by the previous pipeline stage. For each API that should be intercepted, the privacy developer supplies a mock implementation, written in JavaScript, which emulates its expected behavior in a compatible but privacy-preserving way (e.g., a Web Storage API mock would keep all data in memory, while a Fetch

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

Figure 8: The getTrackingId function (Figure 6) naively rewritten. 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. Directly replacing this expression drops the sessionStorage code path, changing the meaning of the code.

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

Figure 9: A high-level illustration of SugarCoat's rewriting strategy, applied to the getTrackingId function from Figure 6. 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.

API mock would return fake responses for network requests). Mock implementations are written once *per API*, not per script, and are reusable and shareable between resource replacements. Privacy developers can optionally specify policies which enable and disable mocks for each target script, controlling the capabilities available to the rewritten versions. Table 1 lists the initial set of Web APIs for which we implemented mocks; see Section 5.2 for a discussion of the scalability of developing additional mocks for SugarCoat.

SugarCoat produces resource replacements in three steps. First, it parses target scripts into abstract syntax trees (ASTs) using the ESPrima[6] JavaScript parser. Then, it rewrites the script ASTs to redirect privacy-relevant API calls to mock implementations. Finally, it transforms the rewritten ASTs into JavaScript source and bundles the source alongside the mock API implementations in a form consumable by off-the-shelf content blocking tools. We describe the most interesting step—the script rewriting—next.

3.3.1 Script AST Rewriting. Given the trace map of source code locations to privacy-relevant Web API calls made at those locations, SugarCoat selectively rewrites the target script ASTs so that the calls are redirected to mock implementations of the same APIs.

A naive approach to rewriting scripts would be to perform an in-place replacement of the exact JavaScript expressions encoding the Web API accesses (e.g., window.localStorage) with expressions that access the mocks instead (e.g., $mockLocalStorage). This, unfortunately, is fragile: Figure 8 shows how this could unintentionally change the meaning of the getTrackingId function from Figure 6 and break compatibility.

Our approach is to work at the JavaScript scope level instead of the expression level. A "scope" here refers to a function body

[6]https://esprima.org

```
Input: scope: top-level scope AST node
       trace: map of code location ⟶ {Web API, ...}
(start, end) ⟵ get source code range covered by scope
trace' ⟵ remove and collect from trace all entries for code locations
  within [start, end]
foreach nested ∈ child scope AST nodes of scope do
    recursively rewrite nested with trace'
end
if trace' is not empty then
    webAPIs ⟵ set of remaining Web APIs in trace'
    insert entry and exit guards for webAPIs into scope
end
```

Figure 10: Scope-narrowing rewriting algorithm. The AST rewriter follows this procedure to insert entry- and exit-guards that redirect Web API calls to mocks.

or the top-level statements in a script; we ignore block scoping. We wrap each scope containing a privacy-relevant Web API call with *entry* and *exit guards*. While control flow is inside a wrapped scope, references in the JavaScript environment to the specific APIs called within that scope are temporarily replaced with their mock equivalents. As shown in Figure 9, the original code is wrapped in a try-finally block; when control flow enters the block, *entry guards* overwrite localStorage and sessionStorage with mocks; when control flow exits the block, *exit guards* restore them. In Appendix A we describe how SugarCoat rewrites different constructs to preserve the scoping of code placed within these try-finally blocks.

This scope-based approach ensures that calls to privacy-relevant APIs can be redirected even when the calls 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, such calls are attributed to the target script most recently pushed onto the stack at the time the call occurs. When a target script calls into a shared library, and that shared library calls a privacy-sensitive API on behalf of the target script, we inject mocks in the calling target script *before* control is transferred to the library and remove them *after* the library returns control to the target script.

Scope-Narrowing Rewriting Algorithm. SugarCoat's AST rewriter is tasked with inserting guards into the AST such that all code locations in the trace map are correctly covered by corresponding guards, while minimizing performance overhead, code bloat from excessive guard insertion, and impact on the rest of the JavaScript environment. To do this, the rewriter follows the algorithm in Figure 10, starting from the AST node corresponding to the top-level script scope and descending recursively into nested function scopes. Each scope "consumes" from the trace map the privacy-relevant code locations between the start and end points covered by the scope's AST node. The rewriter then descends into the scopes nested within the current scope, which in turn "consume" the code locations that belong to them. After traversal, the current scope is left with a list of privacy-relevant code locations for which it is the narrowest, most deeply-nested containing scope. Whenever this list is non-empty, SugarCoat wraps the scope with entry and exit guards corresponding to the Web APIs used at the code locations.

Figure 9 shows how rewriting applies to the tracking.js script from Figure 6. In the original script, references to localStorage and sessionStorage occur in a nested scope: the getTrackingId

Table 2: Summary of Data Gathered to Evaluate SugarCoat

Crawl Configuration	
Measurement period	11/25 – 12/04/2020
Filter lists used	Brave, EasyList, EasyPrivacy, uBlock Origin
# exception rules	6,405
# pages visited	6,195
… embedding an excepted script	999
… which accessed privacy-relevant APIs	902
JavaScript Crawl Statistics	
# scripts loaded	20,981
… matching exception rules	3,034
… rewritten by SugarCoat	1,701
… after deduplication by source code	231
# API calls intercepted by SugarCoat	139,589
… storage API calls	130,494
… network API calls	9,095

function scope, contained within the `initializeTracking` function scope, which is contained in the top-level script scope. Since the `getTrackingId` function scope is the narrowest scope containing the code location, the rewriter selects this scope.

3.3.2 Code Generation and Bundling. As a final step, SugarCoat turns the rewritten ASTs into JavaScript resource replacements. Each resource replacement script is prefixed with mock implementations of the privacy-relevant Web APIs used in the original script (we give a simple Fetch API mock in Appendix B). The rewritten AST is converted to JavaScript code like the sample in Figure 9, and then appended after the mock implementations. SugarCoat packages the resulting source code files into a resource replacement bundle, and generates accompanying EasyList-style filter rules[7] to intercept requests to the original scripts and redirect them to the resource replacements. The output can be dropped into any compatible content blocking tool, such as uBlock Origin [19], AdGuard [2], or the Brave Browser's `adblock-rust`[8] engine.

4 EVALUATION

We evaluate SugarCoat across three dimensions: privacy, compatibility, and performance.

4.1 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 [13] 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.

4.1.1 Web Page Selection. We started by collecting the URLs of Web pages containing privacy-harming scripts which would have been blocked by filter lists, but are excepted for compatibility. We generated this dataset in several steps.

First, we produced a list of Web page URLs from popular sites. We crawled each site in the Alexa 10k, starting from the landing page. For each page, we randomly selected a same-site link on the page (i.e., a link pointing to another page on the same eTLD+1). We repeated the process a maximum of four times, yielding a maximum of five page URLs per Web site, including the landing page.

Second, we randomly sampled Web pages from this list, looking for pages which included at least one excepted script. Specifically, we checked the network requests made by each page against the most popular filter lists (Table 2) using the `adblock-rs` library[9]. Our crawler visited 6,195 pages in total and found 999 pages which included at least one excepted script.

Third, we further reduced this set of pages to those where an excepted script accessed at least one privacy-relevant API (Figure 7). We did so by re-crawling the remaining pages in a PageGraph-enabled browser. From the PageGraph recordings of each page, we identified which scripts access which APIs, and which scripts bring additional "downstream" scripts into the page (e.g., by injecting `<script>` tags). For each re-crawled page, we looked for any instances of privacy-relevant APIs being accessed either (a) directly, by an excepted script or (b) indirectly, by a "downstream" script injected by an excepted script. We found that 902 of the 999 pages (90%) contained at least one such instance, and marked the relevant scripts as "target" scripts for rewriting with SugarCoat.

These pages loaded 20,981 scripts. Of these, 1,701 matched all criteria for rewriting; after deduplicating by checking for scripts with matching source code, we were left with 231 unique target scripts. Table 2 summarizes the properties of the crawl data.

4.1.2 Measurements of Selected Pages. We visited each of the collected 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:

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

In all three conditions, all scripts other than the target scripts were allowed or blocked as normal according to the original filter list rules.

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 source text, and the resource replacements generated by SugarCoat.

Finally, we revisited each of the 902 pages in an unmodified browser (without PageGraph), under each of the three conditions above. We used this additional crawl to measure the performance of

[7]https://github.com/gorhill/uBlock/wiki/Static-filter-syntax#redirect
[8]https://github.com/brave/adblock-rust
[9]https://www.npmjs.com/package/adblock-rs

Table 3: Script-Level Quantitative 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.

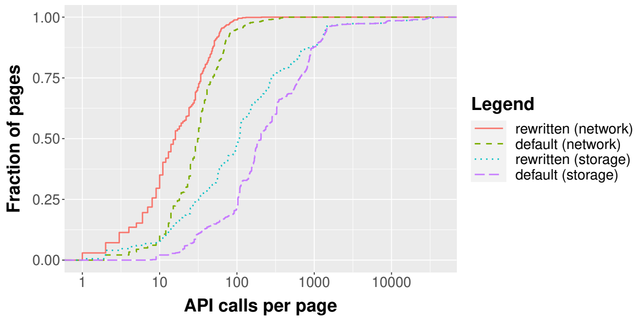

Figure 11: CDFs of the number of storage and network JavaScript APIs called on 902 pages, when privacy-affecting scripts are excepted and run normally ("default"), and when they are replaced with SugarCoat-rewritten versions ("rewritten").

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 blockers in "stock", unmodified Web browsers.

4.2 Privacy Evaluation

We measure how effectively SugarCoat removes privacy-relevant behaviors from target scripts (and their downstream dependencies), and the overall privacy impact of using SugarCoat-generated resource replacements on a page.

4.2.1 Script Level Evaluation. We measure how effectively SugarCoat improves privacy at the script level, comparing the recorded behavior of the original versions of the excepted, Web-compatibility-critical scripts in our evaluation dataset with the behavior of the rewritten versions. Specifically, we count the number of storage and network API calls (Table 1) made by each original script and its rewritten counterpart. As Table 3 shows, SugarCoat dramatically reduces the number of privacy-affecting behaviors scripts engage in.

4.2.2 Page Level Evaluation. We assess the page-level impact of SugarCoat by measuring the differences in privacy-relevant behaviors between "default" and "rewritten" pages (§4.1.2). For each of the 902 pages in our dataset, we extract total counts of JavaScript storage and network API calls from the PageGraph data. Figure 11 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 APIs by non-privacy harming (i.e., non-target) scripts is largely unaffected.

We note that this comparison likely *under-emphasizes* the privacy improvements SugarCoat provides. Not all network requests and storage operations 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 nevertheless allowed to load (generally to avoid breaking compatibility). 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.

4.3 Web Compatibility Evaluation

Our next measurements confirm that SugarCoat does *not* harm desirable page functionality when providing strong privacy benefits.

4.3.1 Qualitative Compatibility Evaluation. We first evaluated SugarCoat's effect on Web compatibility though a qualitative, double-blind, manual evaluation, adopting the approach from [35]. 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 Web sites to serve as test cases, pulled from Web compatibility issue logs reported to the EasyList filter list project.[10] Each selected site included at least one script that was both labeled as harmful by EasyList and 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: "default", "blocked", and "rewritten", as described in Section 4.1.2.

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 Web site was tested independently by two of the six evaluators. We instructed each evaluator to interact with the site for one minute under each of 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 "blocked" 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 blocked 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 blocked or the rewritten configuration.

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

(1) There was no perceptible difference between the configuration presented and the default, control configuration.

[10]The EasyList project notes fixes to compatibility issues by prefacing those git commit messages with "P:" - see https://github.com/easylist/easylist/commits.

Table 4: Qualitative Web Compatibility Evaluation

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

Comparison of how often 6 evaluators considered a Web site working, on a scale of 1 (working) to 3 (broken), when blocking a privacy-harming but compatibility-critical script, and when using a SugarCoat-rewritten version of the script.

(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 summarizes our results. The human evaluators reliably reported their experience of the test Web sites as broken or degraded in some way when the target scripts were blocked, ranging from missing content to non-functional user interface elements to a total failure to load most of the page, with a mean rating of 2.86. When target scripts were instead redirected to SugarCoat-generated resource replacements, evaluators reliably reported their experience as normal, with a mean score of 1.03.[11]

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

We lastly note that we did not collect or process private data or identifiable information from our human evaluators. And since the testing does not qualify as human subject research, we did not seek IRB review. This is consistent with prior work using roughly the same evaluation set-up [20, 22, 35].

4.3.2 Quantitative Compatibility Evaluation. We conducted a quantitative evaluation of SugarCoat's compatibility impact, by comparing the aggregate behavior of "blocked" pages to "rewritten" pages. Using the PageGraph data collected from the "default", "blocked", and "rewritten" versions of each page, 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.[12] 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.

The results of this quantitative evaluation support two 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. Second, using SugarCoat to rewrite 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 eliminates compatibility issues.

Table 5: Quantitative Web 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 API call count 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*.

4.4 Performance

Finally, we describe the performance characteristics of SugarCoat—both the performance of the resource replacements that SugarCoat generates, and the performance of the SugarCoat pipeline itself.

4.4.1 Resource Replacement Performance. We measured the performance overhead of SugarCoat-generated resource replacements used with a "stock", unmodified Chromium-based browser with pre-existing support for content blocking and resource replacement. We visited each page in the 902-page evaluation dataset with "default", "blocked", and "rewritten" filter rule sets loaded into the browser (§4.1.2). We used APIs provided by Puppeteer to extract standard page performance metrics for these visits, covering JavaScript memory usage and the timing of key page-load events.

Table 6 summarizes our results. We observed an average 9% file size increase for the source code of SugarCoat-rewritten scripts compared with their original versions. Part of this comes from the insertion of mock API implementations at the beginning of each rewritten script. This mock code is only inserted once for each intercepted API the script uses, so it represents a bigger relative size increase for smaller scripts than larger ones, and scripts which use a wider variety of privacy-relevant APIs will see more mock code added by the rewriter. The rest of the size increase comes from

[11]In three instances, evaluators rated the SugarCoat condition as a 2 rather than a 1; however, in each of these cases, we observed that the justification given for these ratings was inaccurate (e.g., mistakenly identifying a UI element as missing). To avoid biasing the study, we did not "correct" these ratings. For each test site on which this occurred, the other reviewer assigned to the site rated the SugarCoat version as a 1.

[12]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: Performance Evaluation of SugarCoat-Generated Resource Replacements

Filter Rule Set	**Default**	**Blocked**	**Rewritten**
Target Script Code Size	158 kB (σ = 230 kB)	—	172 kB (σ = 231 kB)
JavaScript Memory Usage (Entire Page)			
Heap Used Size	11 MB (σ = 8 MB)	9 MB (σ = 7 MB)	10 MB (σ = 8 MB)
Heap Total Size	14 MB (σ = 13 MB)	12 MB (σ = 12 MB)	13 MB (σ = 13 MB)
Performance Event Timing (Entire Page)			
DOM Content Loaded	1,607 ms (σ = 995 ms)	1,431 ms (σ = 928 ms)	1,573 ms (σ = 1,019 ms)
DOM Interactive	1,549 ms (σ = 1,004 ms)	1,398 ms (σ = 930 ms)	1,527 ms (σ = 1,023 ms)
Load Event	3,063 ms (σ = 2,410 ms)	2,560 ms (σ = 2,330 ms)	3,025 ms (σ = 2,470 ms)
First Paint	1,026 ms (σ = 709 ms)	842 ms (σ = 674 ms)	960 ms (σ = 838 ms)
First Contentful Paint	1,101 ms (σ = 710 ms)	963 ms (σ = 801 ms)	1,029 ms (σ = 852 ms)

Measurements comparing page performance in a typical browser equipped with each of three filter rule sets: "default", with target scripts excepted; "blocked", with filter rules altered to block the target scripts entirely; and "rewritten", with filter rules inserted to redirect the target scripts to SugarCoat-generated resource replacements. All measurements are medians, taken over the 902-page dataset described in Section 4.1.

Table 7: Performance Evaluation of the SugarCoat Pipeline

Time	Median	σ
Behavioral Profiling (per page)	4.631 ms	11.731 ms
JavaScript Parsing (per script)	34.924 ms	37.068 ms
AST Rewriting (per script)	42.024 ms	32.659 ms
JavaScript Generation (per script)	16.156 ms	24.288 ms

Pipeline benchmarking data gathered from running SugarCoat on a 64-bit, 8-core, Intel i7-6700K system running Ubuntu Linux 18.04.

the insertion of code to intercept and redirect calls that the script makes to the mocked Web APIs. As described in Section 3.3.1, this insertion happens at the JavaScript function level, so multiple calls to an API within a given function do not require any more added code than a single API call. However, the more functions that access mocked APIs, the more interception code is added by the rewriter.

We noticed a small (1-9%) improvement in standard performance metrics when pages were loaded with filter lists configured to redirect target scripts to SugarCoat-generated resource replacements (the "rewritten" condition) compared with filter lists configured to except the target scripts and allow the original versions to load (the "default" condition). Used and total JavaScript heap memory reported by V8 dropped by a megabyte on average. Moreover, key events which mark different points in the browser's page loading and rendering process, commonly used to benchmark user-perceptible Web page performance [14, 15], completed an average of 22–72ms earlier. This is not surprising: mocked APIs, which generate fake results, do less work than the browser-native implementations, outweighing the overhead of redirecting API calls to those mocks. For example, the network API mocks return immediately instead of invoking the network stack; the storage API mocks skip writing data to disk.

Blocking the target scripts further improved page performance (10–18%). However, the scripts we targeted for this evaluation are specifically excepted from blocking by filter list authors to maintain compatibility with the Web pages that embed them. Our qualitative and quantitative compatibility evaluations (§4.3) support this: though blocking these scripts may make the pages run faster, one can expect key functionality to be missing or broken.

4.4.2 Pipeline Performance. To evaluate the performance of the SugarCoat pipeline itself, we benchmarked it on our 902-page evaluation dataset. Tracing API call sites, parsing the input JavaScript, rewriting the ASTs, and generating final resource replacement JavaScript all take milliseconds to complete. We emphasize that these steps are done "offline", by privacy developers generating resource replacements for later use by end users in content blocking tools. As such, our only concern is whether using SugarCoat to generate replacements is computationally prohibitive. As Table 7 shows, this is not the case, and resource replacements can quickly be generated by even a modestly powerful machine.

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 Compatibility Measurements. Because compatibility is a subjective evaluation, it is difficult to measure compatibility with the techniques common to privacy research. Section 4.3 provides two very different attempts to measure compatibility, but any attempt to quantify a subjective measure will be incomplete. 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. Our evaluation assumes that filter lists accurately distinguish privacy-harming resources from benign resources. We also assume that filter lists accurately identify cases where filter rules break Web sites, and that filter list authors address such situations with exception rules. While we know that these assumptions are not always correct, we believe this generalization is useful. First, this assumption is in line with much existing research that treats filter lists as an imperfect-but-best-possible source of ground truth (e.g., [4, 5, 16, 22]). And second, the fact that many millions of people use these lists suggests, even if only anecdotally, that the lists are accurate enough to be useful. Nevertheless, we note this assumption as a limitation, and suggest establishing high-quality ground truth here 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 that arise from content blocking, we note some limitations in our approach. Most significantly, SugarCoat relies on dynamic analysis when tracing (or concretizing) API calls, and so carries with it all the limitations of dynamic analysis systems, including only being able to understand code paths that execute during observation. This limitation can be partially addressed by exercising more code paths (e.g., through human interaction, fuzzing, or guided discovery). SugarCoat can integrate data from multiple observation sessions; our implementation can even map traces gathered from (future) observation sessions of the rewritten scripts themselves back to locations in the original script code, and incorporate this data to iteratively expand the code coverage of the rewritten scripts. However, these techniques only lessen the limitation: they do not fundamentally solve it.

Second, 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 implementations of the relevant APIs before the call sites, and then remove the mock API implementations when control flow leaves the scope. While in practice this approach keeps the API modifications local to the call site, 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 that executes after the target script calls the function, but before the promise resolves, will also see the mock API implementation, regardless of its location in the application. 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 trade-off and limitation for completeness, one which we may address in a future version of SugarCoat.

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 in the Section 5.3.

5.1.4 Adversarial Environments. SugarCoat is designed to improve filter list content blocking, and adopts the same attack model as all filter list blocking. Determined attackers could circumvent the tool's protections, but in practice the cost of circumvention is high enough to make circumvention relatively uncommon, and so make the defense useful in practice, even if not fundamentally robust. This attack model is common to most, if not all, tools that make trust determinations based on URLs. Attackers wishing to evade SugarCoat's protections would do so using the same techniques they would use to evade existing filter list tools (e.g., changing the resource's URL). SugarCoat provides powerful new capabilities to the defender, without adding new circumvention capabilities to the attacker. In short, we intend SugarCoat to advance the state-of-the-art in practical privacy tools, and to deliver real-world privacy improvements, even if those protections are circumventable by particularly determined adversaries.

We also note briefly some of the constraints that make circumvention of filter lists (and so SugarCoat too) costly to attackers. For example, attackers (i.e., trackers) want to make their services easy for Web sites to include, so that non-expert developers can include the tracking scripts. This need for easy deployment reduces the countermeasures available to attackers. Similarly, attackers typically do not control the Web sites fetching their code, which makes it costly and difficult for attackers to update URLs once they have been identified by a filter list. There are many other such practical-though-not-fundamental constraints on attackers; we intend these as illustrative examples for why the attacker model used by SugarCoat and other filter list tools is, in practice, more favoring of the defender than it might first appear.

5.2 Mock Implementation

For this work, we instrumented the Web APIs listed in Table 1. We selected these APIs for two reasons. First, they are privacy-affecting APIs commonly used by tracking scripts to harm privacy. Each of the selected APIs can be used to either store or transmit unique identifiers. Second, we selected these APIs because they represent a range of different interface patterns used in Web APIs (e.g., text parsing for `document.cookie`, synchronous data access for `localStorage`, and asynchronous access for `XHLHttpRequest`), and so demonstrate SugarCoat's flexibility and broad applicability.

SugarCoat's design is general, and can be applied to any properties, methods, or structures in the Web APIs with only small-to-moderate additional development effort. Additional APIs can be instrumented with a one-time cost, and so could be crowd-sourced, upstreamed, and shared. Instrumenting additional APIs does not affect the system's algorithmic complexity.

SugarCoat was intentionally designed to allow developers to instrument new APIs without needing to be familiar with Chromium, Blink, V8, or SugarCoat internals. This is similar to "plugin" or "module" systems in many large, complex projects, allowing developers to extend the system through simple, well-defined interfaces. Instrumenting a new API in SugarCoat requires the following steps:

(1) Implement in JavaScript a mock version of the target Web API property or method. Mocks of many Web APIs already exist, and can be used with small to no modifications (see Appendix B).
(2) Indicate to PageGraph the new JavaScript properties or interfaces being instrumented, by adding a `TrackInPageGraph` annotation in the relevant WebIDL files (see Appendix C).
(3) Bind the new implementation to the method annotated in WebIDL with a small JSON file (instructing SugarCoat that, e.g., `window.fetch` should be redirected to the mock in `fetch.js`).

5.3 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 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. To maximize the benefits of SugarCoat on resource-constrained devices, we need to consider alternative deployment strategies.

Resource-constrained devices may choose to still pre-fetch resource replacements, but to only do so for some threshold of popular

Web sites. Clients could fetch resource replacements as needed from a central repository, and use private information retrieval schemes to avoid leaking browsing habits. Alternatively, replacement scripts could be distributed as compact patches applied at runtime to the original scripts. Each of these strategies allows the client to trade coverage, performance, and privacy against each other, according to the capabilities of the end-user device.

5.4 Other Applications

In this work we focused on using SugarCoat to generate resource replacements to improve the Web compatibility of content blocking tools. Our techniques, however, could be used server-side, or by application developers. For example, with few exceptions, the Web platform does not offer a way to include a script in a page while restricting its capabilities. A Web site author may wish to include some third-party library, but prevent it from calling the network. For many authors, 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 in 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. In this section we discuss how our work relates to existing research in these areas.

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. [26] 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 studied attempts by trackers and advertisers to circumvent or deter filter list use [21, 30, 42, 45], though Iqbal et al. [21] found that filter lists are often effective at defeating these countermeasures (e.g., anti-adblock scripts). Storey et al. [40] 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. [36] found that most rules in the most popular lists provide no benefit. Similarly, Alrizah et al. [3] found popular 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 trackers by behavioral signatures, and Iqbal et al. [22] used lists to train a classifier-based blocker. Other works have used filter lists to train systems that block ads based on their visual appearance [1, 32, 40].

Platform-Wide Web Privacy Improvements. Filter lists improve Web privacy by deploying defenses against known bad actors. A different line of work aims to improve 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. [34] early on documented the benefits of blocking third-party storage altogether. 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 limits cross-site tracking by partitioning third-party storage per first-party, per browser-session [13]. The Tor Browser Bundle and recent versions of Firefox implement similar partitioning strategies [14]. Jueckstock et al. [23] recently demonstrated and evaluated a variant strategy, partitioning third-party storage per first-party, per site-session, to further limit certain forms of cross-session tracking.

A related vein of work focuses on platform-wide (instead of actor-specific) defenses against browser fingerprinting. Nikiforakis et al. [29] proposed a system for preventing fingerprinting based on randomizing the values of some Web APIs. Laperdrix et al. [24] improved the Privaricator proposal by using steganographic-like techniques to minimize the impact of randomization on user-benefiting functionality. Olejnik et al. [31] found that feature removal can be an effective privacy-preserving technique, for features that are rarely used and have a high potential for abuse.

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. [27] and Choudhary [7] both proposed systems for detecting whether a page works correctly when executing in different browser engines. Deursen et al. [41] designed a system for detecting broken applications from differences in traversable links.

Other works have used manual evaluations of sampled Web sites to determine whether a page is working correctly. Snyder et al. [35] used repeated, double-blind manual evaluations of Web sites to determine whether a given Web API was necessary for a page to function correctly. Iqbal et al. [20] used manual evaluations of Web pages to determine whether anti-fingerprinting protections broke pages, and Jueckstock et al. [23] used a similar technique for measuring the impact of different browser storage polices on compatibility. Yu et al. [44] used a novel, indirect method of detecting compatibility issues, by measuring how often users disabled their privacy tools.

Language-based Confinement Techniques. Dynamic information flow control systems like FlowFox [9] and JSFlow [18] can be used to prevent privacy-sensitive data flow leaks. 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 [28]) require heavy modifications to browsers.

On the opposite side of the spectrum, static information flow and static analysis techniques do not translate well to the highly dynamic JavaScript language. Still, our work is similar in spirit to Chugh et al.'s [8], which performs static analysis to make dynamic information flow control practical. 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., [33]), program partitioning (e.g., SIF [6]), and policy weaving (e.g., [17]).

[13]https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/
[14]https://wiki.mozilla.org/Security/FirstPartyIsolation

Systems like BFlow [43] and COWL [39] help developers write applications that do not leak sensitive data, but require significant modifications to applications. Unfortunately, we cannot expect the developers of ad and tracking scripts to do this work for us.

7 CONCLUSION

Content blocking is an important method for protecting privacy, performance, and user agency on the Web, so much so that some government security agencies recommend content blocking tools to government employees to prevent some forms of attack [38]. Unfortunately, Web sites increasingly put content blocking tools in no-win situations: continue protecting user privacy but "break" an increasingly large number of sites, or maintain compatibility by allowing the privacy harm.

We present SugarCoat, an automated, practical system that can shift the state of the Web back in favor of content blocking tools—and, in turn, back in favor of users. SugarCoat provides a solution to these no-win situations: maintain privacy and compatibility through the programmatic generation of privacy-preserving resource replacements. Our tool is intended for real-world use, and is designed to be compatible with existing popular content blocking tools. Brave is already using SugarCoat to generate resource replacements bundled with the Brave Browser, and we are actively working with the maintainers of popular content blocking tools so they can also enhance the privacy of their users' browsing.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Daniel Dubois. This work was supported by the NSF under grant numbers CCF-1918573 and CAREER CNS-2048262, by a gift from Brave Software, and by an NSF Graduate Research Fellowship.

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 *USENIX Annual Technical Conference (USENIX ATC)*.
[2] AdGuard. 2021. AdGuard. https://adguard.com/.
[3] Mshabab Alrizah, Sencun Zhu, Xinyu Xing, and Gang Wang. 2019. Errors, Misunderstandings, and Attacks: Analyzing the Crowdsourcing Process of Adblocking 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. 2021. 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. 2021. EasyList. https://easylist.to/easylist/easylist.txt.
[11] Famlam Fanboy, MonztA and Khrin. 2021. 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. 2020. Tools for Web Developers: Puppeteer. https://developers.google.com/web/tools/puppeteer/.
[14] Google. 2021. Chrome User Experience Report. https://developers.google.com/web/tools/chrome-user-experience-report.
[15] Ilya Grigorik. 2019. Measuring the Critical Rendering Path. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/measure-crp.
[16] David Gugelmann, Markus Happe, Bernhard Ager, and Vincent Lenders. 2015. An Automated Approach for Complementing Ad Blockers' Blacklists. In *Privacy Enhancing Technologies Symposium (PETS)*.
[17] 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.
[18] 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.
[19] Raymond Hill. 2021. uBlock Origin. https://github.com/gorhill/uBlock.
[20] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. 2020. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. *arXiv preprint arXiv:2008.04480* (2020).
[21] 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
[22] 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.
[23] 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).
[24] 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.
[25] 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
[26] 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
[27] Ali Mesbah and Mukul R Prasad. 2011. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*. 561–570.
[28] 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.
[29] 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.
[30] 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
[31] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. 2015. The leaking battery. In *Data Privacy Management, and Security Assurance*. Springer, 254–263.
[32] Adblock Plus. 2018. Sentinel - the artificial intelligence ad detector. https://adblock.ai/.
[33] 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.
[34] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. 2012. Detecting and defending against third-party tracking on the web. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[35] 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.
[36] Peter Snyder, Antoine Vastel, and Ben Livshits. 2020. 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* 4, 2 (2020), 1–24.

[37] Brave Software. 2020. PageGraph. https://github.com/brave/brave-browser/wiki/PageGraph.
[38] Tim Starks. 2021. CISA tells agencies to consider ad blockers to fend off 'malvertising'. https://www.cyberscoop.com/ad-blockers-security-nsa-dhs-wyden/.
[39] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[40] 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).
[41] 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.
[42] 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
[43] 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.
[44] 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.
[45] 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)*.

A REWRITING CHALLENGES

Our AST rewriting pass takes care to handle special cases in the JavaScript language. When we wrap a scope with entry and exit guards, we place its code inside a `try` block and the exit guards inside an attached `finally` clause, as illustrated by Figure 9, to ensure that the exit guards still run if the wrapped code throws an exception. If the scope is the top-level script scope, we must ensure that any top-level declarations we move inside the `try` block still propagate out to the global scope, so that other scripts may access them as they would normally.

Variables declared with `var` are automatically *hoisted* through block scopes in JavaScript, so they require no special handling:

```
try {
  var foo = 'bar';
} finally {}
console.log(foo); // 'bar'
```

Variables declared with `let` and `const`, however, are *block-scoped*:

```
try {
  let foo = 'baz';
  const bar = 'quux';
} finally {}
console.log(foo, bar); // ReferenceError
```

SugarCoat manually hoists these to top-level `let` declarations:

```
let foo, bar;
try {
  foo = 'baz';
  bar = 'quux';
} finally {}
console.log(foo, bar); // 'baz quux'
```

Function declarations in *strict mode* are also block-scoped:

```
'use strict';
try {
  function foo() {
    console.log('bar');
  }
} finally {}
foo(); // ReferenceError
```

SugarCoat manually hoists these to top-level `var` declarations, and preserves function-declaration-order independence by moving all function declarations to the start of the wrapped code:

```
'use strict';
var foo;
try {
  foo = function foo () {
    console.log('bar');
  };
} finally {}
foo(); // 'bar'
```

Finally, class declarations are block-scoped:

```
try {
  class Foo {
    constructor () {
      console.log('bar');
    }
  }
} finally {}
new Foo(); // ReferenceError
```

SugarCoat hoists these to top-level `let` declarations:

```
let Foo;
try {
  Foo = class Foo {
    constructor () {
      console.log('bar');
    }
  };
} finally {}
new Foo(); // 'bar'
```

B EXAMPLE MOCK API

For each API that should be intercepted, the privacy developer supplies a mock implementation, written in JavaScript, which emulates its expected behavior in a compatible but privacy-preserving way. Below we give an example of a mock Fetch API.

```
const fetch = async (url, init = null) =>
  { throw new TypeError('Failed to fetch'); };
exports.fetch = {
  value: fetch, writable: true,
  configurable: true, enumerable: true
};
```

C EXAMPLE ANNOTATED WEBIDL

Our extended version of PageGraph can track accesses to arbitrary Web APIs. Tracking is enabled for a Web API by adding a `TrackInPageGraph` annotation to the corresponding WebIDL code that defines its interface. Below we give an example of annotated WebIDL code from the IndexedDB API, with the added annotation highlighted.

```
[
    ImplementedAs=GlobalIndexedDB
] partial interface Window {
    [TrackInPageGraph]
    readonly attribute IDBFactory indexedDB;
};
```