

Unbundle-Rewrite-Rebundle: Runtime Detection and Rewriting of Privacy-Harming Code in JavaScript Bundles

Mir Masood Ali
mali92@uic.edu

University of Illinois Chicago

Chris Kanich
ckanich@uic.edu

University of Illinois Chicago

Peter Snyder
pes@brave.com
Brave Software

Hamed Haddadi

h.haddadi@imperial.ac.uk
Imperial College London, Brave Software

ABSTRACT

This work presents Unbundle-Rewrite-Rebundle (URR), a system for detecting privacy-harming portions of bundled JavaScript code, and rewriting that code *at runtime* to remove the privacy harming behavior without breaking the surrounding code or overall application. URR is a novel solution to the problem of JavaScript bundles, where websites pre-compile multiple code units into a single file, making it impossible for content filters and ad-blockers to differentiate between desired and unwanted resources. Where traditional content filtering tools rely on URLs, URR analyzes the code at the AST level, and replaces harmful AST sub-trees with privacy-and-functionality maintaining alternatives.

We present an open-sourced implementation of URR as a Firefox extension, and evaluate it against JavaScript bundles generated by the most popular bundling system (Webpack) deployed on the Tranco 10k. We measure the performance, measured by precision (1.00), recall (0.95), and speed (0.43s per-script) when detecting and rewriting three representative privacy harming libraries often included in JavaScript bundles, and find URR to be an effective approach to a large-and-growing blind spot unaddressed by current privacy tools.

1 INTRODUCTION

An enormous body of research has established Web content filtering (e.g., blocking advertising, tracking, and other unwanted network requests on websites) as an important and effective technique for improving privacy[36, 48], security[42, 66], and performance[35, 53]. Most Web content filtering approaches rely on crowd sourced lists of regular-expression-like rules that describe which URLs the browser should load, and which should be blocked.

This approach—broadly, URL based content filtering—works because URLs in practice provide useful and stable information about the resources they map to. In some cases this is because of the text in the URL (e.g., browsers can make a reasonable guess about the purpose of JavaScript returned from a URL like `https://advertising.example/tracker.js`), or because experts have manually evaluated the resource returned from a URL and found it to be similarly harmful to users.

However, modern Web development practices make URL based content filtering increasingly difficult. Previously, Web applications were often delivered as a collection of discrete JavaScript files, each fetched independently from their own URL (e.g., `/script/library.js`, `/script/tracker.js`, `/script/app.js`), which allowed URL-based content filtering tools to easily block some parts of an

application, but not others. Increasingly though, developers integrate bundling tools as part of their build and deployment practices, compiling all of the libraries and application code into a single file unit, which is delivered to the browser from a single URL (e.g., `/script/bundle.js`).

These bundling approaches, inadvertently or otherwise, circumvent URL based content filtering tools. When applications are delivered as a single bundled code unit, URL based filtering tools can no longer block just parts of the application; blocking a site’s JavaScript becomes an all-or-nothing proposition. And since blocking all JavaScript on a page breaks useful functionality on many sites, in practice content filtering tools are reduced to blocking nothing, reintroducing the privacy, security, and performance issues the user wanted to avoid in the first place.

This work presents the design and implementation of Unbundle-Rewrite-Rebundle (URR), a system to enable content blocking in modern Web applications, even when Web applications are deployed as a compiled, single file JavaScript bundle. In other words, URR aims to enable browsers to avoid executing the code from `/script/tracker.js`, while still executing the non-privacy harming code originally provided in `/script/library.js` and `/script/app.js`, *even when* all three libraries are bundled and delivered in `/script/bundle.js`.

URR is a novel, practical solution to a problem that has been explored by a vein of related Web privacy research. Works like [19] identify that bundled applications are widespread and pose a serious challenge to Web privacy, and [59] found that blocking these bundled JavaScript resources often broke the benign, desirable parts of websites. Systems like [60] showed that bundled applications could be automatically rewritten to prevent privacy harm, though with expensive precomputation, which rendered practical deployment prohibitive. URR is a first-in-class approach to solving the privacy and security harms caused by bundled JavaScript applications, in a method that is performant and practical.

To do so, URR solves several non-trivial challenges:

First, the system must identify known privacy harming code (e.g., the code delivered from `/script/tracker.js`) within the larger bundled application, in the absence of any information about the URL the code originally came from. This identification must be robust even across the kinds of code modifications and transformations JavaScript bundlers make in their build processes (e.g., minification, dead-code elimination, tree-shaking).

Second, URR must remove known-privacy-harming code from the bundled application, without breaking desirable functionality

in the surrounding code. Just deleting unwanted libraries from a bundled application will, in practice, be counter productive, since the application will fail when trying to access now-deleted functions and classes defined by the deleted library. A useful solution must remove the unwanted, target libraries from the bundled application *without breaking surrounding code*.

Third, such a system must be performant, and be able to Debundle, analyze, modify, and reconstitute bundled Web applications at runtime, and quickly, in a way that maintains the usefulness of the Web application. If the performance overhead of a privacy-and-security preserving system is too costly, then the system is in practice unusable, and so not meaningfully useful at benefit Web users.

URR is implemented in several parts: i. as a database of signatures of ASTs of real world known-privacy-harming code, ii. a library of crowd sourced privacy-preserving alternative implementations of privacy-harming code, designed to remove privacy harming behaviors without impacting surrounding application code, iii. a browser extension that, at runtime, decomposes a bundled JavaScript application into its constituent libraries, detects the sub-ASTs from the bundled application that come from known-privacy-harming libraries, and rewrites the bundled application with the stub-libraries in place of the privacy-harming versions.

More broadly, this work makes the following contributions to Web privacy and security.

- (1) A novel algorithm for efficiently generating fingerprints of bundled modules within JavaScript libraries. These fingerprints are robust to many of the modifications that bundling tools make during their compilation process (e.g., label minification, “tree-shaking”).
- (2) An open sourced, empirically tuned system for:
 - (a) unbundling applications into their constituent libraries
 - (b) generating fingerprints for each sub-library and checking them against a database of known unwanted JavaScript libraries
 - (c) replacing unwanted JavaScript libraries with compatibility-preserving “shim” implementations, which maintain the library’s API “shape”, while removing any privacy-or-security affecting behaviors
 - (d) reconstituting the resulting new application into a new bundle, that can then be passed to the browser’s JavaScript engine for normal execution.
- (3) An empirical evaluation of the accuracy and performance of our system when applied to a representative crawl of the Web, finding that our system results in libraries being blocked on 7% of the top 10K sites in the Tranco list, within practical performance bounds.
- (4) An open source implementation of our system as a Firefox extension, along with the complete dataset for all discussed figures and measurements.

2 BACKGROUND

This section first provides a primer on concepts relevant to bundling and their use in web development. It then presents a simple example that highlights the limitations of existing content blocking

approaches. The section concludes by outlining the properties of an effective solution.

2.1 Bundles and Relevant Concepts

As websites and web applications grow more complex, they require a plethora of functionality, often aided by numerous libraries and dependencies. Handling these dependencies can prove to be quite strenuous, especially when needing to take into account the range of platforms on which the code needs to correctly execute. JavaScript bundlers are essential tools used by web developers to streamline the handling of code and dependencies within complex web applications. At its core, a JavaScript bundler is a utility that gathers and wraps code from multiple JavaScript files. Bundles not only reduce the number of network requests required to load a web page but also optimize the handling of dependencies and various aspects of development and production environments. The simplicity of the example hides certain nuances necessary to understand the complexity of bundles. Below, we introduce a few fundamental concepts that can help better understand JavaScript bundles.

2.1.1 Modular Programming. Web developers design and create websites in different environments than browsers. These environments have their own caveats and follow different programming concepts and philosophies. Modular programming is a general programming concept where developers separate complex application code into independent pieces called *modules*. A module forms the atomic unit of a bundle, and roughly corresponds to a code snippet relevant to a file or library that exports functionality that is consumed by other modules. Popular JavaScript development environments like Node.js [1] adopt a modular programming approach. Developers can create their own modules and use reuse modules available in the npm registry [2].

2.1.2 JavaScript Module Systems. Like development environments and programming approaches, JavaScript module systems are also not a monolith. Depending on the context in which they are consumed and executed, JavaScript modules express functionality in different ways. The two most popular module systems for JavaScript are (1) the ECMAScript modules (ESM) which are consumed with `import` statements, and (2) CommonJS (CJS) modules which are consumed with `require` statements. While Node.js supports both types of modules, web browsers only recognize `import` statements. Bundles therefore include wrappers around modules and provide workarounds for `require` statements within web browsers.

2.1.3 Inter-module Dependencies. Larger projects include multiple libraries and packages, and as a result comprise numerous, inter-dependent modules. When bundles gather and parse all the modules that need to be combined, they create a dependency graph that helps determine (1) the order and chain of dependencies between modules, and (2) which code snippets can be combined within a module and which snippets need to be split across multiple modules.

2.1.4 Minification. Bundlers additionally perform a code transformation step that reduces the overall size of the code. Depending on various configuration and optimization options, minification

```

1 (function (modules) {
2   // The module cache
3   var installedModules = {};
4
5   // The require function
6   function __webpack_require__(moduleId) {
7     // Check if module is in cache
8     if (installedModules[moduleId]) {
9       return installedModules[moduleId].exports;
10    }
11    // Create a new module (and put it into the
12    // cache)
13    var module = (installedModules[moduleId] = {
14      exports: {},
15    });
16
17    // Execute the module function
18    modules[moduleId].call(
19      module.exports,
20      module,
21      __webpack_require__
22    );
23
24    // Return the exports of the module
25    return module.exports;
26  }
27
28  // Load entry module and return exports
29  return __webpack_require__(0);
30 })([
31   /* 0 */
32   function (module, exports, __webpack_require__) {
33     const hello = __webpack_require__(1);
34     console.log(hello.sayHello("Webpack"));
35   },
36   /* 1 */
37   function (module, exports) {
38     exports.sayHello = function (name) {
39       return "Hello, " + name + "!";
40     };
41   },
42 ]);

```

Listing 1: A non-minified example of a webpack bundle.

returns an irreversible code output containing randomized, unidentifiable variable names and changes to the code syntax that only retains its underlying logic.

2.1.5 Source Maps. During development, bundles provide source maps as a key to reverse minification and debug parts of code. These files help map minified code back to their unminified counterparts, and hence identify individual modules. However, source maps are not available by default in production, making it difficult to reverse engineer bundled code in the wild.

2.1.6 Popular Bundlers. Examples of popular JavaScript bundlers include Webpack [5], Browserify [8], Rollup [6], and Parcel [7], each of which offer unique features and advantages. In this work, we focus on Webpack because it is the most popular and mature JavaScript bundler as determined from GitHub stars [3], NPM weekly downloads [4], and prior work [54].

2.1.7 General Structure of Webpack Bundles. Listing 1 presents an example of a script comprising a webpack bundle. We describe the example below.

- Webpack wraps the bundled code in an **Immediately Invoked Function Expression (IIFE)**. As a result, the bundle is executed as soon as it is loaded, thereby making all necessary functions and variables available in the global scope.
- The bundle comprises a modular system where each module is represented as a function in an **array or object of modules**. Listing 1 contains a module array in L30-42.
- **Function wrappers around each module** handle dependencies on other modules and gather any variables and functions exported by the module.
- The `__webpack_require__` function (Listing 1, L6-26) loads and executes modules and **manages exports and dependencies**.
- The bundle executes an **entry point module** (module 0 in Listing 1). The entry point module uses `__webpack_require__` to load functionality from other modules (Listing 1, L33).

Overall, bundlers allow developers to organize their code into modules, manage dependencies, and apply various optimizations like minification and code splitting, resulting in more efficient, maintainable, and faster-loading web applications.

2.2 Motivating Example

In this section we present an example that shows how typical content blocking approaches work and why they are ineffective against bundles.

2.2.1 Generic Code Inclusion. Consider the toy example presented on the left half of Figure 1. Here, the website loads two scripts, each of which define global functions. The first script, `setup.js`, loads from the website’s domain itself and contains benign code relevant to the website’s functionality. The second script, `track.js`, loads from a known tracking domain, `tracker.com`, and returns a script that creates a unique identifier to track the user. When the website includes the two scripts from two separate `<script>` tags, it triggers two network requests, one to `website.com` and another to `tracker.com`.

2.2.2 Typical Content Blocking Approach. A typical approach to blocking privacy-harming scripts involves the use of a curated list of domains and regular expressions (for example, from EasyList [24], EasyPrivacy [25]). These lists are developed from manual contributions and include domains and paths to known privacy-harming resources. Content blocking tools (e.g., Adblock [26], uBlock [37]) pull from these filter lists. The tools intercept outgoing network requests, compare them against entries in filter lists, and create interventions if they find a match. In the toy example, a filter list contributor might add the rule `||tracker.com/track.js`. Thereafter, when the website creates two network requests, the content-blocking tool permits a request to `script.js` but blocks the request to `tracker.com/track.js`. This way, the privacy-harming script is neither loaded nor executed in the user’s browsing session.

2.2.3 Limitations of Existing Content Blocking Approaches. Consider the scenario presented in the right half of Figure 1. Unlike the previous example, the website includes a single `<script>` tag that fetches code from its own server. The resulting network request does not have a corresponding entry in the filter list and is therefore not blocked by the content blocking tool. The server responds with a bundled script that includes both, benign code (`setup()`)

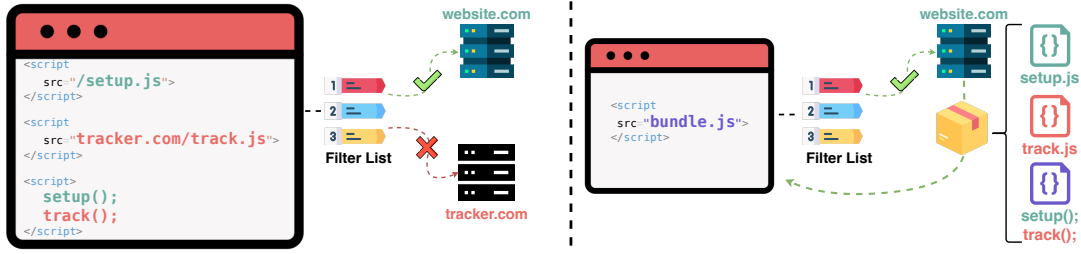


Figure 1: Motivating example.

and privacy-harming code (`track()`). Bundled resources expose multiple shortcomings of existing content-blocking approaches, which we briefly describe below.

- Privacy-harming code can be fetched from **multiple, variable resource paths**, including from first-party domains, making it impossible for contributors to manually detect and curate a comprehensive, non-exhaustive list of entries.
- Bundles include code from multiple resources. As a result, privacy-harming code is embedded along with necessary and benign functionality. Blocking the network request itself can break the website. Existing approaches need to adopt an approach that targets **embedded resources** instead of the entire resource itself.
- Bundles also **mutate code included in modules** with wrappers and through various minification and obfuscation techniques. These mutations make it difficult to identify content with comparisons against regular expressions.

2.3 Properties of an Ideal Solution

We outline the properties of a general solution to identifying and replacing privacy-harming modules from bundled scripts.

First, a robust solution should be able to **identify and target specific modules**. Unlike typical content blocking approaches, the solution cannot rely on domain-based blocking or attempt to replace the a script in its entirety.

Second, as a corollary to the previous point, the solution should have **limited impact on benign functionality**, i.e., the solution should limit the effect of its intervention to privacy-harming code, leaving execution of other modules untouched. This also involves ensuring that any dependencies on the targeted module are handled in a way that limits side-effects.

Third, the solution should be **generalizable across multiple dimensions**. It should be applicable to multiple privacy-harming libraries (and multiple versions of libraries) that are of interest to content blocking tools. Additionally, it should also be generalizable to different bundling strategies and robust against minification and obfuscation techniques.

Finally, the solution should have a **limited performance overhead**. While the solution executes in real-time, i.e., as scripts are loaded and executed in the browser, it needs to limit its effect on the usability of websites.

3 UNBUNDLE-REWRITE-REBUNDLE DESIGN

Unbundle-Rewrite-Rebundle (URR) adopts a static analysis approach that leverages the code structures of privacy-harming modules to identify and neutralize them when embedded in bundled scripts without disrupting the functionality of other components of the application.

URR adopts a four-step process (see Figure 2):

- First, URR generates an Abstract Syntax Tree (AST) representation of a given script. It then analyzes the structure to identify if the script is a bundle that comprises one or more modules. If so, URR gathers the component modules for further analysis.
- Next, URR processes each module into an implementation agnostic representation, i.e., stripping variable names, function names, and object properties. It creates a bottom-up hash of the AST structure and uses the resulting representation for comparison.
- Next, URR compares each processed module against previously generated representations of privacy-harming modules. If URR finds a match, it marks the corresponding module for replacement.
- Finally, URR replaces each marked privacy-harming module with a corresponding, benign replacement. In doing so, URR ensures that access and use of the replacement does not break other parts of website functionality. It then stitches the bundled script back together and moves it along for consumption and execution.

All the steps mentioned above are performed in real-time, i.e., when resources are loaded in the user’s browser. However, processed representations of target modules and benign replacements are created and gathered offline.

3.1 Gathering Modules

URR first generates an Abstract Syntax Tree (AST) representation of a JavaScript resource. The AST representation helps gather the syntactic features of the script and provides insight into the structure of the code.

Thereafter, URR uses the AST representation to determine two aspects of the loaded script. First, URR determines if the script comprises a bundle that includes one or more modules. Second, URR gathers the sub-trees corresponding to each of the identified modules.

Below, we describe the webpack-specific implementation for this phase.

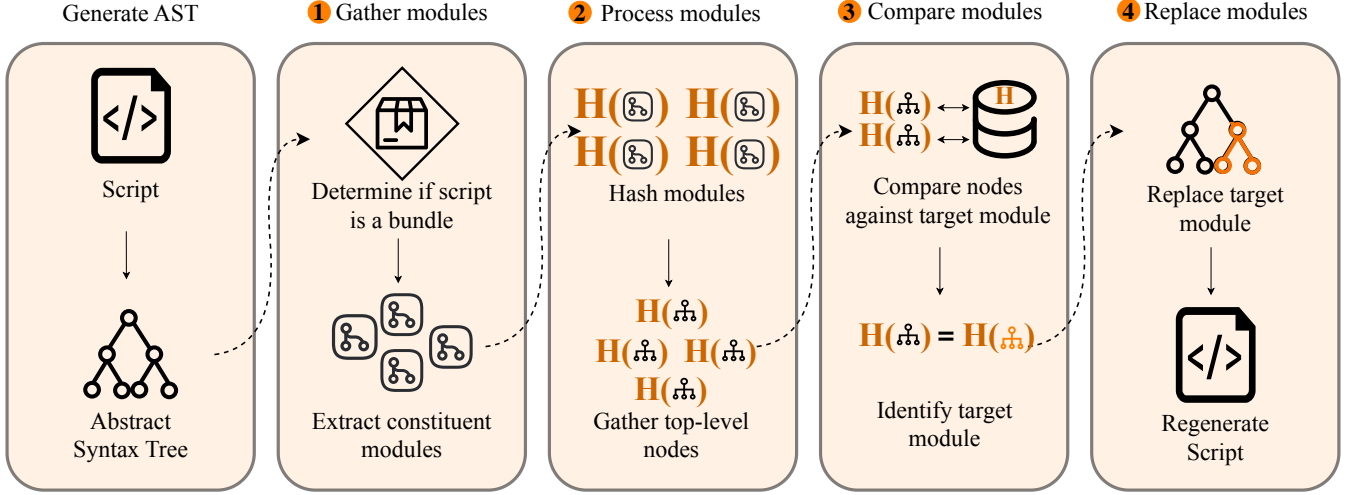


Figure 2: Overview of the framework.

Table 1: Refining code to identify bundled webpack modules within scripts.

	Round 1 (n = 288)			Round 2 (n = 432)			Round 3 (n = 614)		Spot Check (n=300)	
	Manual	Code Initial	Code Refined	Manual	Code Initial	Code Refined	Manual	Code	Manual	Code
Annotation 1: Webpack Bundle										
# Webpack Bundles	104	153	111	156	162	163	207	217	99	102
Precision		0.68	0.94		0.94	0.96		0.95		0.97
Recall		1	1		1	1		1		1
Annotation 2: Component Modules										
# Modules	6,857	7,847	6,857	9,816	9,156	9,090	12,889	12,913	5,701	5,690
Precision		0.87	0.99		0.99	0.99		0.998		0.998
Recall		1	1		0.93	1		1		0.996

3.1.1 Gathering JavaScript resources. In order to create a valid parser for bundles in the wild, we gathered examples of resources loaded in popular websites. We developed a puppeteer-based crawler that, upon visiting a page, intercepts network requests and stores a copy of observed script responses. We visited domains from the Tranco list [41] and gathered 30,930 scripts from 1,063 sites (1K crawl).

3.1.2 Generating ASTs. For each of the gathered scripts, URR uses acorn, a community-developed, open-source JavaScript parser to generate an AST [17]. The generated AST complies with the ESTree Spec to ensure a consistent, standardized representation that can be reproduced by alternative implementations [44].

3.1.3 Code Development and Refinement Methodology. We began with an understanding of the general structure of the output of a webpack bundle (see Listing 1). We developed a script that parses an AST and looks for a module array or object, i.e., an array or object that comprises functions. We used this initial logic and adopted a

code optimization methodology based on ground-truth gathered through expert manual annotation. We describe the process below.

- (1) First, we randomly sampled 100 scripts (without replacement) from the JavaScript resources gathered during the 1K site crawl.
- (2) We manually evaluated the plaintext script and the AST of each sampled resource. In doing so, we checked for the presence of webpack-specific code patterns. We annotated each resource with a boolean value indicating whether we determined the script as a webpack bundle (**Annotation 1**). The manual checks included:
 - (a) identification of code pattern similar to webpack’s specific function that handles dependencies (see Listing 1, L6-26);
 - (b) identification of code patterns specific to webpack chunks, i.e., files comprising webpack modules separate from the entry point bundle;
 - (c) identification of objects and arrays specific containing functions with parameters, exports, and return statements similar to webpack’s function wrappers;

Algorithm 1 Processing modules.

Input: AST \leftarrow module AST
Output: hashedAST \leftarrow processed representation of the AST
procedure HASHMODULE(*node*)
 $childrenHash \leftarrow 0$
 for each *child* in *node.children* **do**
 $childrenHash \leftarrow childrenHash + HASHMODULE(child)$
 end for
 return $hash(node.type + childrenHash)$
end procedure

- (d) keyword searches that indicate the use of webpack in the use of webpack in the creation of the resource.
- (3) For any resource annotated as a webpack bundle, we also annotated the resource with the number of component modules we identified (**Annotation 2**).
 - (4) We repeated Steps 1-3 until at least we had annotated 100 scripts as webpack bundles.
 - (5) We then executed our code to automatically analyze and annotate each sampled resource in a similar manner, i.e., (a) **Annotation 1**: whether the script is a bundle, and (b) **Annotation 2**: the number of component modules identified in the script.
 - (6) We compared the code-generated annotations against the manual annotations and gathered the set of differences. For each incorrectly labeled script, we analyzed the code outputs and refined its logic. Refinements included addressing edge cases, handling multiple IIFEs in a single resource, etc.
 - (7) We used the refined version of our code to annotate the same sample and noted any improvements. This refined version of the code as the initial version of the subsequent phase, against new samples.
 - (8) For each subsequent round, we repeated Steps 1-7, appending the existing sample with 50 manually annotated webpack resources (Step 4). We stopped refining our logic when we observed negligible improvement in the precision, recall, and accuracy of code-generated annotations between consecutive phases.

We repeated the process for three rounds. Table 1 presents the numbers and metrics for both annotations and the associated metrics for each round of refinement. At the end of the third round, our code identified a given script as a webpack bundle with 95% precision and gathered modules with 99.8% precision.

We used the resulting logic to annotate all scripts gathered in our 1K crawl. We annotated a total of 11,995 scripts as webpack bundles. Finally, we performed a spot check of the generated annotation. We randomly sampled and manually annotated 300, previously unsampled scripts of which we identified 99 scripts as webpack bundles. Upon comparing and verifying the code-assigned annotation, we observed a 97% precision rate in identifying webpack bundles and a 99.8% precision rate in gathering component modules.

3.2 Processing Modules

Each module gathered from the previous phase comprises a subtree, i.e., a partial AST of the larger AST representation of the script. The module’s AST contains information about both, the structure

of the underlying code and associated names of variables, functions, and properties as included within the script. Script attributes like variable names are extremely volatile and have limited use in reliably identifying target modules. URR therefore only considers (1) the *structure* of the AST, i.e., the parent-child relationships between the nodes that comprise the AST, and (2) the *type* of each AST node, i.e., attributes which comply with the ESTree Spec, and are hence limited to a deterministic set of values.

To distill and represent only these specific attributes from a module’s AST, we adopt a version of a cryptographic concept used in integrity verification: Merkle trees [47]. As an example, consider a list of data blocks that need to be verified (e.g., transactions in a blockchain). We represent this list of blocks in a tree structure. Consider a tree where each leaf node represents a piece of data, and each non-leaf node is a cryptographic hash of its child nodes. These hashes propagate upwards, converging into a singular root hash, known as the Merkle root. Any alteration in the foundational data triggers a modification in the root hash, instantly signaling tampering or modifications. However, if the root hash is verified, all of its children are verified as well.

URR adopts a similar methodology, briefly presented in Algorithm 1. Given a module’s AST, it performs the following processing steps.

- (1) URR begins traversing the module AST from the root node.
- (2) For every *child node*, it recursively calls the function to gather the *child node*’s hash. The hash of each child in-turn results from the hashes of its children.
- (3) It sums the hashes of all *child nodes* of the root node.
- (4) To generate a hashed representation of the root node, URR concatenates the *type* of the node to the sum of hashes of its children, and hashes the resulting string.
- (5) It returns the processed representation of the AST.

This processed representation now comprises an alternative tree representation with equal depth to its original counterpart. However, tree-based comparisons are complex and add a large performance overhead. Recall that while processing an AST, top-level nodes contain information about underlying children. Therefore, comparing a limiting comparison to a few high-level nodes of the processed module can provide insight into the similarity of the module’s AST. To this end, URR gathers a list of top-level nodes, i.e., given a module with webpack’s function wrappers, it gathers a list of root nodes of each subtree corresponding to a top-level statement within the function. Additionally, URR associates each entry in this list with a weight that represents the number of child nodes it represents, i.e., it weighs a node representing a complex function higher than a node representing a simple variable declaration.

3.3 Comparing Modules

URR uses the list of top-level nodes to compare against a database of top-level nodes of processed module representations corresponding to targeted libraries. While this database is generated offline, processed modules for these libraries are also gathered in a similar manner to the logic presented in Algorithm 1. We provide details regarding the creation of this database in Section 4. Here, we focus our discussion on the comparison and identification of processed modules.

Algorithm 2 Comparing modules against target libraries.

Input:
 $TOP_LEVEL_NODES \leftarrow$ list of top-level nodes of module
 $DATABASE \leftarrow$ hashes of known libraries
 $CANDIDATE_LIBRARIES \leftarrow$ empty set
Output:
 $MATCH \leftarrow$ weight of the associated match
for each $nodeHash$ in TOP_LEVEL_NODES **do**
 $librariesWithThisNode = DATABASE[nodeHash]$
 for each $[library, weight]$ in $librariesWithThisNode$ **do**
 $CANDIDATE_LIBRARIES[library] += weight$
 end for
end for
 $bestMatch = \max(CANDIDATE_LIBRARIES)$
if $bestMatch = targetLibrary$ **then return** $CANDIDATE_LIBRARIES[targetLibrary]$
end if

Algorithm 2 presents an overview of the comparison strategy. We present a brief description below.

- (1) URR traverses the list of hashes of the top-level nodes that represent the module observed in the wild.
- (2) It looks up the hash of each node in the database and identifies every library that has the same top-level node. It considers any such library as a *candidate library*.
- (3) It appends the weight of the node for each *candidate library* to previous matches, if any. Therefore, the weights associated with a *candidate library* increases each time it includes a match.
- (4) Once all top-level nodes have been traversed, URR identifies the *candidate library* with the highest associated weight.
- (5) If the highest match corresponds to the target library, URR returns the weight of this match, thereby marking the module as a candidate for replacement.

3.4 Replacing Modules

URR replaces each marked privacy-harming module with a corresponding, benign replacement. It can perform this action in one of two ways; (1) URR can replace the module’s AST with an alternative, benign AST and then regenerate the script from the modified AST; (2) Alternatively, URR can identify the string indices for the module within the textual representation of the script, and place the benign replacement between these indices. Regardless of the approach it adopts, URR ensures that access and use of the replacement does not break other parts of website functionality.

Replacements for targeted modules are manually created. Given the nuances of each target library and its eventual, mutated representation in webpack bundles, we describe the three categories of replacements to consider within each module.

- **Global Variables.** If libraries expose global variables on the window object, these variables need to be made available. Additionally, the types of such variables need to be retained.
- **Exports.** The replacement needs to ensure that values previously exported from the target module remain available. When the module is consumed by other modules, they need to be made accessible, even if replaced with a benign version.

- **Webpack-based Function Wrappers.** When webpack wraps modules in functions, it passes specific parameters included in the function signature. Additionally, values exported by modules are appended as properties to webpack objects. Replacements need to ensure that these parameters and exports work seamlessly.

Finally, each replacement needs to ensure the consistency of variable types, i.e., functions be replaced with functions, constants be replaced with constants, and so on. We further provide examples of specific replacements in Section 4.

4 EVALUATION

In this section we evaluate the effectiveness of URR in identifying specific libraries in bundled scripts captured in the wild. We select three libraries to target and describe the process of gathering their module representations for comparison. We then use URR to identify target libraries in scripts gathered from a crawl of popular websites. Next, we develop a deployment of URR as a Firefox extension and evaluate the performance overhead introduced by its interventions. We conclude by describing the creation and evaluation of benign replacements.

4.1 Evaluation Dataset

We first describe the selection of example libraries that we use to identify in the wild. We describe the offline process of gathering representations for these libraries which URR can then use to identify libraries in the wild.

4.1.1 Target Library Selection. We select three libraries for our evaluation —FingerprintJS [31], Sentry [58], and Prebid [51]. Each of these libraries are included in filter lists [24, 25] used by popular content blocking tools [26, 37]. Additionally, all three libraries provide the option for developers to use their npm packages with bundled code [10, 12, 14].

FingerprintJS [33] is a popular browser fingerprinting library that tracks and identifies users. When embedded within a website that a user visits, the library performs various client-side operations that query browser attributes, and stores a unique identifier for the user. Besides its use in user authentication [43] and ad fraud prevention [9], the library de-anonymizes sensitive user activity across sites. Since the domains that the library uses are blocked by popular content blocking tools, FingerprintJS recommends that developers use their npm package or self-host a copy of their scripts [32].

Sentry [34] is an analytics tool that offers libraries for performance and error monitoring. The library helps developers gathers details about user interaction, DOM events, console logs, and network calls. The library lets developers decide how they use the information gathered from users and offers multiple integrations, including with multiple third-party analytics services [56]. Since the CDNs that host by Sentry code are blocked by popular content blockers, the library recommends that developers get around this restriction by bundling Sentry’s npm package into their app [57].

Prebid [52] is a popular advertising library that developers can use to add header bidding to their application. The library integrates with numerous advertising, analytics, and user tracking libraries, providing support for bidding on targeted advertisements. Similar to previously discussed libraries, Prebid is included within filter lists and content blocking tools. The library is open source and

Table 2: An overview of of the build options used to gather representations for the target library.

Build Options	Possible Values
Module Systems:	
↪ Dependency Statement	[import, require]
Webpack Optimizations:	
↪ usedExports	[true, false]
↪ concatenatedModules	[true, false]
Minifier Options:	
↪ passes	[0, 1, 2]
↪ arrows	[true, false]
↪ dropConsole	[true, false]
↪ unsafeCompress	[true, false]
↪ unsafeMethods	[true, false]
↪ unsafeUndefined	[true, false]
↪ unsafeArrow	[true, false]
↪ unsafeCompare	[true, false]
↪ typeofs	[true, false]
Browser Support:	
↪ ie8	[true, false]
↪ safari10	[true, false]

is available to be added to bundles with an npm package. Prebid provides specific instructions on integration with webpack [20, 51]. The library needs to be compiled with Babel and makes use of specific plugins when loaded with webpack. It provides specific instructions for webpack configurations, which gives us insight into its use when integrated by bundles in the wild.

4.1.2 Bundler Configurations. Web applications are developed in a large variety of environments, frameworks, and configurations, before they are compiled and shipped to production. As a result, when the same npm package is bundled within different applications, its modules will generate a wide range of AST structures. These variations make it difficult to gather a definitive AST representation for the package that will return perfect matches when compared against scripts found in the wild. However, the AST of the module corresponding to each library is complex and hence, unique. Our intuition is that the AST of a given library will be closer to differently configured ASTs of the same library than to ASTs of other libraries.

To gather multiple options for module representations and help us get the closest match to a target library in the wild, we gather a list of popular build options to be considered for each library. Table 2 These options cover four aspects that we describe below.

- (1) **Module Systems.** Depending on the target npm package and the development environment of the web application, the library may be consumed as a CommonJS module or an EcmaScript module. We gather configurations for both module types.
- (2) **Webpack Optimizations.** When bundling code from multiple files, webpack keeps track of imported and exported values with the help of a dependency graph. Depending on the module system, the output environment, and developer’s configurations, it provides options to automatically identify and discard unused parts of code and to reduce the size of the final output.

The use of these options can change how the module appears in the output bundle.

- (3) **Minifier Options.** Once webpack creates a bundle output, it uses Terser [63] to minify the code. Developers can provide additional options to ensure that the output works with their intended production environment, ensuring that their applications work in all supported environments. We consider a list of compression options that alter the syntax of the module in the bundled output.
- (4) **Browser Support.** Developers can additionally specify support for legacy browsers. These options override other minification and compression to ensure that the output bundle is also compatible with legacy browsers.

We gathered all combinations of the build options and created a total of 24,576 build configurations that we then used to bundle each target library.

4.1.3 Creation of Target Module Representation Database. We gather various hashes for these libraries with the help of a large set of versions and different webpack configuration options, resulting in multiple AST representations for each target library. We briefly describe the steps below.

- (1) **Library Versions.** We gather a list of past versions released by the library on the npm registry. We download and install each version in a separate barebones application.
- (2) **Barebones Application.** For each version, we create a barebones Node.js application with a single JavaScript file. The file consumes the target library with either an import or a require statement.
- (3) **Build Options.** For each library version, we build multiple webpack bundles, one for each combination of build options.
- (4) **Target Module.** When building a bundle, webpack provides the option to gather relevant info about individual modules [64]. We use this information to identify the file, name, and position of the module corresponding to the target library. Thereafter, we gather an AST representation of the output bundle and extract the sub-tree corresponding to the target module.
- (5) **Processing AST.** We process the extracted AST in a similar manner to the steps described in Section 3.2. We then gather a list of hashes for the top-level nodes from each bundle along with their corresponding weights.

4.1.4 Additional Library Configurations. In addition to the three target libraries, we gathered a list of 10k popular npm packages based on their download counts [27, 39, 50]. From this list, we collected a random sample of 10 versions each of 1k libraries. For each sampled library version, we generated 100 webpack bundles with each with a different, random build option. We extracted and processed their modules and gathered a weighted list of hashes corresponding to the top-level nodes of their ASTs.

4.2 Replacements

Prior work has extensively studied and determined the creation and use of non-breaking, benign replacements for privacy-harming code [60]. We follow similar principles and extend example replacements used by uBlock Origin [15].

4.2.1 Creating Benign Versions of Target Libraries. For each of the three libraries, we considered associated source code and documentation to create equivalent benign versions. We briefly describe them below.

FingerprintJS, when loaded as a library from its npm package, provides a function, `load()`, which returns a Promise that resolves to a computed `visitorId`. While the computation includes privacy-harming code, the benign replacement that we developed only returns a randomly generated `visitorId`. We used a similar approach to the existing replacement included within `uBlock` origin [16], but modified the same to a Node.js module.

Sentry provides a browser library as an entry point npm package [14] to include its functionality within web applications. The library further includes other related packages and integrations that vary across developer and application use cases. We used the official documentation as a guideline and developed replacements for exported values, included those imported from other Sentry libraries. For each class that the library exports, we developed an equivalent benign class with a constructor that accepts any argument it is passed. Similarly, we developed benign function replacements, with no return value, for each exposed function. For functions that return promises, we resolve into an empty promise. Finally, since the library additionally makes functionality available globally, via the `window` object, we also make the benign version of the library globally available.

Prebid provides a similar npm package to `sentry`, i.e., its main functionality is included from an entry point module, while developers can make use of additional integrations. We focus on the entry point module, i.e., `pbjs`, since all other functionality is accessed from this module. We follow a similar approach, i.e., creating benign replacements for each function and variable made available on the entry point module, the Prebid API for publishers (i.e., web application developers), and ensure that any access to the benign version is considered valid and does not throw an error when accessed.

4.2.2 Gathering Bundled Versions. For each replacement module, we created a barebones application that accesses the library functionality. The barebones application helps evaluate each replacement, ensuring that the application remains functional. We package the resulting application and gathered the webpack module corresponding to the benign replacement, for use in URR’s deployment.

4.3 Framework Effectiveness

In this section, we describe our evaluation of URR’s effectiveness in identifying target libraries within scripts captured in the wild.

4.3.1 JavaScript Resource Dataset. We used a puppeteer-based crawler that visited sites from the the Tranco list [41] and gathered JavaScript resources through network interception. We previously used this dataset to create a bundle parser, described in Section 3.1. The dataset comprises 30,930 scripts from 1,063 sites, of which 11,995 scripts are bundled resources.

4.3.2 Gathering Matches. For each bundle in the dataset, URR extracted and processed component modules (see Algorithm 1). It then gathered a weighted list of top-level nodes from the processed module. URR then compared each module against all libraries in the evaluation dataset (see Section 4.1). For each module, we noted

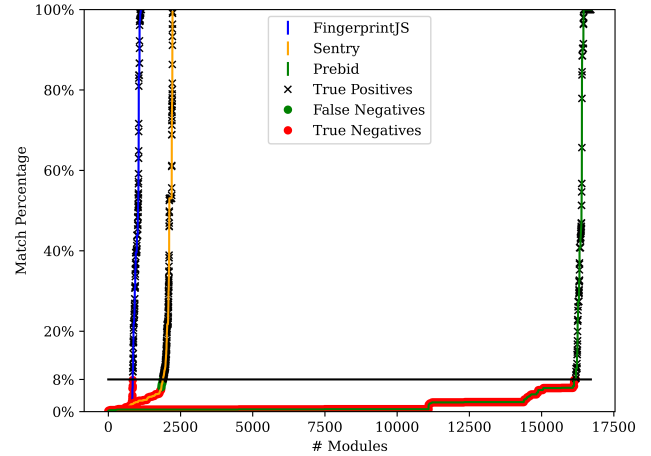


Figure 3: An overview of the target libraries identified from a crawl of the Tranco 10K. In addition to the 697 instances of true positive (x) matches above the 8% threshold, we observed >15K instances of true negative (•) matches of modules below the threshold.

the library in the evaluation dataset with the closest match. Finally, we gathered a list of modules for which the closest match was one of the three target libraries. We gathered a total of 73 matches for FingerprintJS, 324 matches for Sentry, and 3,532 matches for Prebid.

4.3.3 Manual Verification. For each match, we manually verified the code embedded between the corresponding indices in the script. We observed the code structure, the use of specific properties, and the number and types of exported variables, functions, and objects. We then manually annotated each match as a true positive or a false positive.

4.3.4 True Positive Threshold. We observed the tradeoff between precision and recall for the matches returned for all three target libraries. We looked for the lowest threshold match percentage for an AST for which all three libraries observed 100% precision, i.e., any AST that returned a positive match above this threshold was correctly annotated. We therefore arrived at a common match threshold value of 8%. Despite observing a small drop in recall for Sentry, we prioritized precision to ensure that URR never blocks benign modules even if it misses some privacy-harming scripts.

4.3.5 Evaluation on a larger crawl. Next, we performed a larger crawl of the web, gathering scripts from the top 10K sites in the Tranco list. We used URR to evaluate these scripts and employed the $\geq 8\%$ match threshold discussed earlier. We detected FingerprintJS on 205 sites, Sentry on 213 sites, and Prebid on 325 sites. Overall, URR identified at least one of the three target libraries on 7% ($n = 697$) of the top 10K sites. Figure 3 presents an overview of our observations.

4.4 Performance

In this section, we provide a sample deployment of URR as a Firefox extension. We use this non-optimized deployment to evaluate an

Table 3: Time taken (ms) by the the extension to process scripts using network interception.

Phase	# Scripts	Time (ms)		
		μ	σ	Median
Buffering script text	6,702	690.69	1,189.26	240
Gathering modules	6,699	46.17	98.25	9
Processing modules	2,135	326.31	635.58	114
Comparing modules	2,135	1.15	2.28	0

upper bound on the latency introduced by different phases of the pipeline and their effect on page load times.

4.4.1 Firefox Extension. Considering URR’s ability to recognize bundles given the entire contents of scripts, we determined that an approach that intercepts network requests and responses will be a useful initial deployment. Considering Chrome’s restrictions on content filtering from extensions [23], we developed a Firefox extension that intercepts and modifies the content of responses across all domains that the browser visits [65].

4.4.2 Crawl. We evaluated the performance of a browser instance with and without the extension by visiting sites in the Tranco list [41]. For each web page, we performed the following steps.

- (1) First, we created and initialized a new browsing profile.
- (2) Next, we used Mozilla’s web-ext [11] to load a Firefox instance with the web extension.
- (3) We used puppeteer [13] to connect to the browser instance, and visit the web page in a new tab and wait for 30 seconds.
- (4) The extension’s background script intercepts and evaluates all script-based network requests. It captures and stores the time taken by each step in the script evaluation pipeline [45].
- (5) The extension’s content script captures metrics relevant to web page performance.
- (6) We closed the browser instance and deleted the user profile directory and Firefox’s caches from the filesystem to ensure a fresh browsing state for subsequent visits.
- (7) We repeated steps 1-6 in a browsing profile without our extension installed and recorded relevant web page performance metrics for comparison.

4.4.3 Pipeline Performance. First, we discuss the evaluation of individual scripts. Table 3 shows the time taken by different phases of the pipeline. We observed the gathering modules and comparing processed modules have the least impact, taking on average 46.17ms and 1.15ms respectively. URR takes the longest time to process modules, i.e., recursively compute hashes for all nodes in an AST, which takes 326.31ms on average for each script.

However, we observed that the largest chunk of the time was consumed outside of URR’s evaluation and instead spent in buffering incoming response chunks to load the entire script. This is a limitation of the mode of deployment, i.e., Firefox extensions that modify the body of network responses bypass the browser’s optimized cache for scripts [46]. The deployment waits for large

Table 4: Time taken (ms) to load a page with and without the extension installed, from a crawl of $n = 963$ web pages.

Metric	w/o Extension			w/ Extension		
	μ	σ	Median	μ	σ	Median
First Contentful Paint	2,735.55	1,690.62	2,550	3,029.51	2,236.36	2,423
DOM Interactive	2,796.39	1,687.63	2,620	3,367.55	2,204.51	3,052
Page Load	4,790.39	3,156.72	4,130	7,107.45	4,759.42	6,078

network responses to complete before evaluating the script. In Section 5, we discuss alternative deployments that evaluate code at runtime and can bypass dependence on network load times.

4.4.4 Page Performance. Next, we discuss the effect of the sample deployment on page load. Recall that we used a content script to gather metrics from the page context for each visit with and without the extension. Table 4 presents a snapshot of our observations. We describe three collected measures below.

First Contentful Paint (FCP) is a timing measure that shows when the browser renders the first bit of content. The content could be any text, image, video, canvas, or non-empty SVG. The timing shows the first instance that the user has an indication that the page is loading. We observed that the extension added 293.36ms to the average time a user would wait for such an indication.

DOM Interactive indicates the time taken for the Document Object Model (DOM) parser to finish its work on the main document, i.e., the time taken to construct the DOM. This time can be affected by the parser-blocking JavaScript. Note that the extension’s script evaluation runs outside the main thread. We observed that the extension added 571.66ms on average, i.e., a user would have to wait an additional 0.5s before the browser has parsed the DOM.

Page Load indicates the total time taken for all resources to load. It indicates that network requests for all scripts, images, and other resources have completed. This metric does not indicate actual end-user experience since it depends on device capabilities and various network conditions, but in our case, provides insight into the additional time added by the extension buffering responses and evaluating script contents. We observed that pages loaded with the extension took an additional 2,317.06ms on average before triggering the load event.

Overall, deploying URR as a browser extension slightly increases the time taken to load the page, adding minimal performance overhead and ensuring that users can continue to meaningfully interact with the website.

5 DISCUSSION

5.1 Applicability to Other Systems

In this work we target and evaluate against JavaScript bundles generated by Webpack. We selected Webpack because it is the most popular bundling system on the Web. However, there are many other bundling approaches used on the Web, some variations on the same approach used in Webpack some fundamentally different and even working on different levels of the deployment process. We here briefly discuss these other bundling formats, and how URR could be extended to apply to them (and with what difficulties).

Webpack is generally used to combine multiple different JavaScript libraries and code units, and to process them into a single JavaScript

file, to make deployment easier and (in some cases) execution faster. While Webpack is the most common tool for this purpose, there are many others, for use with different languages, build chains, testing frameworks, and, in some cases, to also bundle additional resource types beyond JavaScript files. Examples of these alternative bundlers include Browserify¹ and Gulp², among many others.

URR could easily be extended to cover these other bundling tools, as at root they all operate in the same manner (i.e., consume multiple JavaScript files, preprocess them, and then generate code for the resulting combined AST). To do so would only require generating new signatures for ASTs of each target privacy-harming library generated by the bundler’s preprocessing and rewriting phases, and understanding the structure of each bundler combines the ASTs generated by each input library into the final, resulting code unit. This is work that would only need to be done once per bundler version, and then could be shared across all URR clients.

Another approach to JavaScript bundling is to directly combine each JavaScript code unit into a single archive, and to ship the entire archive to the client, along side the website’s initial HTML. This approach, exemplified by Google’s WebBundles³ proposal, does not preprocess or otherwise modify the include JavaScript code units; included files are directly copied into the bundle archive. Cloudflare’s Cloudflare’s Managed Components⁴ product can also be seen as a form of this kind of bundling, though instead of all combined into a single archive, they’re instead delivered “on demand”, as managed by an overriding “manager” application, making any URLs unpredictable.

Extending URR to cover these kinds of bundled application would be trivial, since bundled JavaScript files are not modified or preprocessed in the bundling process. Identifying them within the bundled application is straightforward. Similarly, rewriting unwanted code is similarly trivial, since it only requires swapping the original file with the privacy-preserving alternative (either in the original archive, or returned as a new subresource by the “manager” application).

5.2 Alternative Deployment Strategies

In this work we implement URR as a Firefox extension. We choose a Firefox extension because only Firefox’s extension API includes the ability for an extension to buffer and rewrite a fetched subresource⁵ (like a JavaScript file), before the file is seen, parsed, and executed by the JavaScript engine.

However the same approach we take in the extension could be identically deployed from other decision points, either to be more general across browsers (e.g., as part of a man-in-the-middle proxy) or more specific to particular browsers (e.g., as a modification to the V8 parsing pipeline). The majority of the resources needed for URR to work could apply equally across all of these intervention points (e.g., fingerprints of the ASTs of unwanted code, bundle “debundling” logic, privacy-preserving replacement AST subtrees).

¹<https://browserify.org/>

²<https://gulpjs.com/>

³<https://webpack-wg.github.io/bundled-responses/draft-ietf-wpack-bundled-responses.html>

⁴<https://managedcomponents.dev/>

⁵<https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/filterResponseData>

However, each of these intervention points would come with their own largely-predictable trade offs. Analyzing, debundling, and rewriting JavaScript bundles as part of a man-in-the-middle proxy would work for any browser or network tool, though with a performance impact (since JavaScript engine optimizations like partial or lazy compilation would not be possible). Likewise, pushing URR logic into the JavaScript engine directly would likely allow for greater performance, at the cost development and maintenance cost, and requiring browser specific implementations. That said, we reemphasize that most of the novel aspects of URR would be generic and shareable across all possible intervention points.

5.3 Diversity of Target Library Representation

URR requires a significant amount of precomputation to work effectively. Specifically, URR requires precomputing a signature for every AST for each library or code unit that should be rewritten at runtime. While this is a significant improvement over the existing state of the art ([60], for example, requires the precomputing each target *bundled application*, which will both be orders of magnitude larger in occurrence, but also enormously larger in terms of required disk space), its still not trivial. The same target library can give different signatures depending on library version (e.g., FingerprintJS v2 vs v3), bundler version (e.g., Webpack v4 vs Webpack v3), bundler optimization strategies (e.g., “tree-shaking” or no), library integration method (e.g., CommonJS vs ECMAScript modules) among other dimensions.

As discussed in Section 3, URR uses several heuristics to effectively generalize signatures, to flatten the number of dimensions of signatures needed per target library (e.g., label stripping, AST simplification, etc.). Never the less, there is still a tradeoff between coverage—generating as many signatures as possible, to correctly identify the same target code across a wide range of representations—and concision—minimizing the memory, matching time, and disk space used on each URR client at runtime.

5.4 Replacements

While other phases of URR are programmatically generated, we manually created benign replacements. This part requires an understanding of both, the specific bundler and the target library itself. While prior work has shown approaches that can automate this creation, we leave the adaption of a similar approach in the context of bundles as an avenue to explore in future work. Prior work like [60], which allows for the automatic creation of privacy-and-compatibility preserving versions of JavaScript libraries, could be leveraged to greatly expand the number of target libraries URR can identify and rewrite in bundled applications.

6 RELATED WORK

This work contributes to and builds on an enormous body of research relevant to web privacy, tracking, and content blocking. In this section, we highlight existing research that relates to the our framework’s design.

Filter Lists. This research is closely tied to a broad domain of research investigating the advantages, effectiveness, and responses to filter list-based content blocking. Note that existing filter lists (e.g., EasyList [24], EasyPrivacy [25]) are developed manually and

community-maintained. Merzdovnik et al. [48] performed a large-scale study that highlighted the effectiveness of filter list-based browser extensions and also indicated that these tools often lead to a *decrease* in overall CPU usage, even when considering their own overhead. Gervais et al. [36] quantified the privacy gain from ad-blockers and discovered that these tools can reduce interactions with third-party entities by up to 40% with default settings.

However, other studies have pointed out significant inefficiencies in filter lists. Snyder et al. [61] highlighted the abundance of “dead-weight” rules that offer no discernible benefits in popular lists. Similarly, Alrizah et al. [18] found that popular lists contain a large number of false positives which can take two or three months to be discovered.

Automated Content Blocking Approaches. Numerous studies have developed approaches to either help automatically generate filter list rules or develop alternative methods to block privacy harming content. AdGraph [38] created a graph-based machine learning approach that used features like URL length and origin to differentiate between benign and privacy harming resources. Bhagavatula et al. [21] also used URL-based features to train a machine learning model for resource filtering. Chen et al. [22] used filter lists as ground truth and developed behavioral signatures based on the JavaScript event loop, while Sun et al. [62] classified JavaScript execution based on Web API calls. Le et al. [40] developed a reinforcement learning framework that generates filter list rules specific to a site of interest and showed that their approach was comparable in visual breakage to manually-created filter lists.

JavaScript Analysis. Prior work that perform static or dynamic analysis on JavaScript to create “pre-filters” [29] for malicious scripts, detect malicious scripts that camouflage as benign scripts [28], and detect scripts that evade detection by adopting various obfuscation strategies [30]. In their analysis of the Top 10K sites in the Alexa list, Moog et al. [49] found that 90% sites contained a minified or obfuscated script.

More relevant to our work, Rack and Staicu [55] presented a method for detecting and partially reverse engineering bundles. However, unlike the automated approach we present in § 3.1, their approach to detect bundlers requires *manually* curated fingerprints; They manually analyzed the output of a bundler and gathered unique code snippets which they then used to identify whether a script is a bundled resource. Further, their approach to partially reverse engineer bundles relies on the availability of source maps, i.e., a feature used during development mistakenly being exposed in production. As a result, they were only able to extract modules from around 10% of the bundles they detected in the wild. We instead present an automated approach (see § 3.2 and § 3.3) that can reverse engineer bundles regardless of the accidental availability of source maps. However, in support of our choice of bundler, they found Webpack to be the most popular bundler by an order of magnitude. They observed the presence of 1.27 Webpack bundles on average per site, with 0.11 Browserify bundles per site being the closest alternative choice on the top 100K sites.

7 CONCLUSION

Content blocking plays a crucial role in safeguarding privacy, enhancing performance, and preserving user autonomy online. However, the increasing use of bundlers presents a challenge – websites often intermingle tracking code with benign code within a single script – thereby rendering traditional URL-based content blockers ineffective.

We present a framework, Unbundle-Rewrite-Rebundle (URR), that detects bundles and reverse engineers them back to constituent modules. We develop an approach to identify the privacy harming modules that we then replace with benign alternatives. We demonstrate the effectiveness of our system in identifying Webpack bundles and further developed signatures for a fingerprinting library (FingerprintJS), an advertising library (Prebid), and an analytics library (Sentry). Our implementation can identify the bundled versions of these libraries in the wild via a similarity threshold that minimizes false negatives and prevents false positives within our training data. Leveraging our approach, we found the use of these libraries within bundled scripts on 697 sites of the Tranco 10K. Further, we implemented a prototype deployment of URR as a browser extension and observed that the extension only adds 0.5s to the DOM Interaction Time on a page visit.

URR can expand existing content blocking approaches to combat the use of bundlers to hide privacy harming code. It can be further adapted to combat alternative bundling strategies and can be further used to detect the use of other privacy-harming libraries, to bolster web privacy protections.

REFERENCES

- [1] 2009. Node.js. <https://nodejs.org/en/>
- [2] 2010. npm | Home. <https://www.npmjs.com/>
- [3] 2014. Github Repository | webpack. <https://github.com/webpack/webpack>
- [4] 2014. NPM Registry | webpack. <https://www.npmjs.com/package/webpack>
- [5] 2014. webpack. <https://webpack.js.org/>
- [6] 2015. Rollup | Rollup. <https://rollupjs.org/>
- [7] 2018. Parcel - The zero configuration build tool for the web. <https://parceljs.org>
- [8] 2020. Browserify. <https://browserify.org/>
- [9] 2023. Customer Case Studies | Fingerprint Device Intelligence Platform. <https://fingerprint.com/case-studies/>
- [10] 2023. @fingerprintjs/fingerprintjs. <https://www.npmjs.com/package/@fingerprintjs/fingerprintjs>
- [11] 2023. Getting started with web-ext. <https://extensionworkshop.com/documentation/develop/getting-started-with-web-ext/>
- [12] 2023. prebid.js. <https://www.npmjs.com/package/prebid.js>
- [13] 2023. Puppeteer | Puppeteer. <https://pptr.dev/> publisher: Google, Inc..
- [14] 2023. @sentry/browser. <https://www.npmjs.com/package/@sentry/browser>
- [15] 2023. uBlock: Web Accessible Resources. https://github.com/gorhill/uBlock/tree/9123563895f0499849b4d85c4f95e1ed6ace2231/src/web_accessible_resources
- [16] 2023. uBlock: Web Accessible Resources: fingerprint3.js. https://github.com/gorhill/uBlock/blob/9123563895f0499849b4d85c4f95e1ed6ace2231/src/web_accessible_resources/fingerprint3.js
- [17] 2023. acornjs. 2023. acorn: A small, fast, JavaScript-based JavaScript parser. <https://github.com/acornjs/acorn>
- [18] Mshabab Alrizah, Sencun Zhu, Xinyu Xing, and Gang Wang. 2019. Errors, Misunderstandings, and Attacks: Analyzing the Crowdsourcing Process of Ad-blocking Systems. In *Proceedings of the Internet Measurement Conference* (Amsterdam, Netherlands) (IMC '19). Association for Computing Machinery, New York, NY, USA, 230–244. <https://doi.org/10.1145/3355369.3355588>
- [19] Abdul Haddi Amjad, Danial Saleem, Muhammad Ali Gulzar, Zubair Shafiq, and Fareed Zaffar. 2021. TrackerSift: untangling mixed tracking and functional web resources. In *Proceedings of the 21st ACM Internet Measurement Conference* (Virtual Event) (IMC '21). Association for Computing Machinery, New York, NY, USA, 569–576. <https://doi.org/10.1145/3487552.3487855>
- [20] Babel. 2023. The compiler for next generation JavaScript. <https://babeljs.io/>
- [21] Sruti Bhagavatula, Christopher Dunn, Chris Kanich, Minaxi Gupta, and Brian Ziebart. 2014. Leveraging Machine Learning to Improve Unwanted Resource

- Filtering (*AISeC '14*). Association for Computing Machinery, New York, NY, USA, 95–102. <https://doi.org/10.1145/2666652.2666662>
- [22] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. 2021. Detecting Filter List Evasion with Event-Loop-Turn Granularity JavaScript Signatures. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1715–1729. <https://doi.org/10.1109/SP40001.2021.00007>
- [23] Oliver Dunk. 2023. Improving content filtering in Manifest V3. <https://developer.chrome.com/blog/improvements-to-content-filtering-in-manifest-v3/>
- [24] EasyList Authors. 2023. EasyList. <https://easylist.to/easylist/easylist.txt>
- [25] EasyPrivacy Authors. 2023. EasyPrivacy. <https://easylist.to/easylist/easyprivacy.txt>
- [26] eyeo GmbH. 2023. Adblock Plus: The world's #1 free ad blocker. <https://adblockplus.org/>
- [27] Tristan F. 2023. npm-rank: get popular npm packages. <https://github.com/woorm/npm-high-impact>
- [28] Aurore Fass, Michael Backes, and Ben Stock. 2019. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (London, United Kingdom) (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1899–1913. <https://doi.org/10.1145/3319535.3345656>
- [29] Aurore Fass, Michael Backes, and Ben Stock. 2019. JStap: a static pre-filter for malicious JavaScript detection (ACSAC '19). Association for Computing Machinery, New York, NY, USA, 257–269. <https://doi.org/10.1145/3359789.3359813>
- [30] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. 2018. JASr: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.
- [31] FingerprintJS. 2023. Browser fingerprinting library. <https://github.com/fingerprintjs/fingerprintjs>
- [32] FingerprintJS. 2023. Evade ad blockers. https://github.com/fingerprintjs/fingerprintjs/blob/master/docs/evade_ad_blockers.md
- [33] FingerprintJS, Inc. 2023. The device intelligence platform. <https://fingerprint.com>
- [34] Functional Software Inc. 2023. Sentry: Application Performance Monitoring & Error Tracking Software. <https://sentry.io/welcome/>
- [35] Kiran Garimella, Orestis Kostakis, and Michael Mathioudakis. 2017. Ad-blocking: A study on performance, privacy and counter-measures. In *Proceedings of the ACM on Web Science Conference*. 259–262.
- [36] Arthur Gervais, Alexandros Filios, Vincent Lenders, and Srđjan Capkun. 2017. Quantifying web adblocker privacy. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 21–42.
- [37] Raymond Hill and Nik Rolls. 2023. uBlock Origin - Free, open-source ad content blocker. <https://ublockorigin.com/>
- [38] 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)*. 763–776. <https://doi.org/10.1109/SP40000.2020.00005>
- [39] Andrei Kashcha. 2023. npmrnk: npm dependencies graph metrics. <https://github.com/anvaka/npmrnk>
- [40] Hieu Le, Salma Elmalaki, Athina Markopoulou, and Zubair Shafiq. 2023. AutoFR: Automated Filter Rule Generation for Adblocking. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 7535–7552. <https://www.usenix.org/conference/usenixsecurity23/presentation/le>
- [41] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Koczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS 2019)*. <https://doi.org/10.14722/ndss.2019.23386>
- [42] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and Xiaofeng Wang. 2012. Knowing your enemy: understanding and detecting malicious web advertising. In *Proceedings of the ACM conference on Computer and communications security (CCS)*. 674–686.
- [43] Xu Lin, Panagiotis Ilia, Saumya Solanki, and Jason Polakis. 2022. Phish in Sheep's Clothing: Exploring the Authentication Pitfalls of Browser Fingerprinting. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 1651–1668. <https://www.usenix.org/conference/usenixsecurity22/presentation/lin-xu>
- [44] Sebastian McKenzie, Kyle Simpson, Mike Sherov, Ariya Hidayat, Adrian Heine, Dave Herman, and Michael Ficarra. 2023. The ESTree Spec. <https://github.com/estree/estree>
- [45] MDN. 2023. storage. <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/storage/publisher: Mozilla>
- [46] MDN. 2023. webRequest.filterResponseData(). <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/filterResponseData/publisher: Mozilla>
- [47] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 369–378.
- [48] 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. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 319–333.
- [49] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 569–580. <https://doi.org/10.1109/DSN48987.2021.00065>
- [50] npm/registry. 2023. package download counts. <https://github.com/npm/registry/blob/master/docs/download-counts.md>
- [51] Prebid. 2023. A free and open source library for publishers to quickly implement header bidding. <https://github.com/prebid/Prebid.js>
- [52] Prebid.org Inc. 2023. Boost Programmatic Advertising Revenue. <https://prebid.org/>
- [53] Enric Pujol, Oliver Hohlfeld, and Anja Feldmann. 2015. Annoyed users: Ads and ad-block usage in the wild. In *Proceedings of the Internet Measurement Conference (IMC)*. 93–106.
- [54] Jeremy Rack and Cristian-Alexandru Staicu. 2023. Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (Copenhagen, Denmark) (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3576915.3623140>
- [55] Jeremy Rack and Cristian-Alexandru Staicu. 2023. Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (Copenhagen, Denmark) (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 3198–3212. <https://doi.org/10.1145/3576915.3623140>
- [56] Sentry. 2023. Analytics. <https://develop.sentry.dev/analytics/>
- [57] Sentry. 2023. Dealing with Ad-Blockers. <https://docs.sentry.io/platforms/javascript/troubleshooting/#dealing-with-ad-blockers>
- [58] Sentry. 2023. Official Sentry SDKs for JavaScript. <https://github.com/getsentry/sentry-javascript>
- [59] Michael Smith, Peter Snyder, Moritz Haller, Benjamin Livshits, Deian Stefan, and Hamed Haddadi. 2022. Blocked or broken? Automatically detecting when privacy interventions break websites. *arXiv preprint arXiv:2203.03528* (2022).
- [60] Michael Smith, Pete Snyder, Benjamin Livshits, and Deian Stefan. 2021. Sugar-Coat: Programmatically Generating Privacy-Preserving, Web-Compatible Resource Replacements for Content Blocking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (Virtual Event, Republic of Korea) (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2844–2857. <https://doi.org/10.1145/3460120.3484578>
- [61] Peter Snyder, Antoine Vastel, and Ben Livshits. 2020. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. In *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems (Boston, MA, USA) (SIGMETRICS '20)*. Association for Computing Machinery, New York, NY, USA, 75–76. <https://doi.org/10.1145/3393691.3394228>
- [62] Jingxue Sun, Zhiqiu Huang, Ting Yang, Wengjie Wang, and Yuqing Zhang. 2021. A system for detecting third-party tracking through the combination of dynamic analysis and static analysis. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 1–6. <https://doi.org/10.1109/INFOCOMWKSHPS51825.2021.9484564>
- [63] Terser. 2023. JavaScript mangler and compressor toolkit. <https://terser.org/webpack>
- [64] webpack. 2023. Stats Data. <https://webpack.js.org/api/stats/>
- [65] Rob Wu. 2022. Manifest v3 in Firefox: Recap & Next Steps. <https://blog.mozilla.org/addons/2022/05/18/manifest-v3-in-firefox-recap-next-steps/>
- [66] Apostolis Zarras, Alexandros Kapravelos, Gianluca Stringhini, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. 2014. The dark alleys of madison avenue: Understanding malicious advertisements. In *Proceedings of the Conference on Internet Measurement Conference (IMC)*. 373–380.