

Dredging the River Styx: Fortifying the Web through Robust and Real-Time Script Attribution

Kostas Drakonakis*
Technical University of Crete
Chania, Greece
kdrakonakis@tuc.gr

Sotiris Ioannidis*
Technical University of Crete
Chania, Greece
sotiris@ece.tuc.gr

Jason Polakis
University of Illinois Chicago
Chicago, USA
polakis@uic.edu

Abstract—The modern web ecosystem relies heavily on the inclusion of third-party scripts as they offer useful, and often necessary, functionality. This inclusion leads to the “blending” of code from different origins, which has significant ramifications. Specifically, the inability to effectively and robustly disambiguate between first-party and embedded third-party scripts can severely undermine the security and privacy guarantees of existing defenses (e.g., blocking trackers or preventing vulnerabilities such as DOM XSS), as well as the validity of web measurement studies. To address that gap we propose StyxJS, a system that is able to provide real-time attribution of third-party scripts while preventing evasive tactics that can be employed by malicious scripts. This is achieved through an automated pipeline consisting of stack walking, script rewriting, browser API overriding, and tamper-proofing mechanisms. Crucially, our system does not require any developer input or prior knowledge about the website and can, thus, be readily incorporated into any countermeasure or web measurement apparatus that requires robust script attribution. We conduct an extensive experimental evaluation of our system and demonstrate that it accurately captures more script inclusion techniques compared to prior work, while incurring a negligible performance overhead, and effectively maintains page-deployed security mechanisms (e.g., CSP). We also detail the straightforward process and benefits of retrofitting a varied set of existing defenses on top of StyxJS, as well as leveraging it to analyze the web ecosystem. We will release our system as an open source project, to allow security researchers and practitioners to benefit from StyxJS’ capabilities.

1. Introduction

The web is driven by complex relationships and interdependencies of different parties. One of the earliest and most prominent scenarios for such dependencies are *third-party* (3P) scripts, where a *first-party* (1P) site includes JavaScript (JS) files from other domains. These scripts offer a multitude of additional functionalities, ranging from analytics [1] and advertisements [2], to Single Sign-On [3], [4] and fingerprinting [5]. 3P script inclusions have been prevalent for over a decade [6], and recent studies have found that inclusions have become much more prevalent and complicated, as 3P scripts can *implicitly* load additional scripts [7]–[9].

While 3P scripts facilitate development and offer rich functionalities, they come at a significant cost: once loaded, they operate with the *same privileges* as the 1P and can access the same information and functionality, making them seemingly *indistinguishable* at runtime. This is exacerbated by JS’ dynamic nature, allowing scripts to execute further scripts at will [10], [11]. As such, buggy, compromised, or malicious 3P scripts can severely affect websites’ security and privacy posture. For instance, they can introduce client-side vulnerabilities [8], [12], prevent HTTPS deployment [9], carry out cookie-stealing attacks [13] or invasively track and fingerprint users [14]–[18]. Therefore, robust 3P script attribution capabilities are *crucial* for building effective security countermeasures and conducting accurate web security measurements which often rely on analyzing 3P scripts’ behavior. Unfortunately, no well-defined, standardized method for achieving this exists, and prior approaches do not fully achieve the desired results.

In more detail, existing countermeasures and 3P analyses relying on JS instrumentation [7], [12], [19], [20] often rely solely on *naive* stack walking for attribution, which is susceptible to bypasses when handling *dynamically* injected scripts [10], [21]. Another common pitfall is overlooking several dynamic JS-inclusion methods. Systems built with in-browser instrumentation are generally more robust when it comes to attribution, due to the browser engine’s rich, low-level information (e.g., PageGraph [22], ScriptChecker [23]). Unfortunately, such systems suffer from certain fundamental limitations. First, low-level modifications are inherently tied to a single browser version, hindering wide and fast adoption, while also requiring users to build their browser from source to utilize them. Moreover, wide deployment would require adoption by browsers *and* website developers [23]–[26], which rarely occurs in practice [27], further diminishing their practicality. They can also require expensive, *offline* pre-processing [28]. As such, while extremely useful for certain measurements and studies, such systems cannot support real-time privacy and security enhancements for end users. Crucially, we also experimentally demonstrate that bypasses are feasible against such systems as well.

Motivated by the core problem of achieving *robust* and *real-time* 3P script attribution, we develop StyxJS. Our system, realized as a browser extension, operates at the JS layer without requiring any modifications to the browser. StyxJS’ workflow includes stack inspection, on-the-fly script rewriting and JS API overriding, and is able

*. Kostas Drakonakis and Sotiris Ioannidis are also with FORTH.

to capture *all* 3P scripts in a page, including dynamically loaded ones. It achieves this in real-time and without any *a priori* knowledge of the encountered scripts. In addition, carefully designed tamper-proofing mechanisms ensure that our system is robust against evasion. At the same time, our design was guided by extreme precaution for respecting security mechanisms deployed by web apps. Moreover, due to its plugin-based design, StyxJS provides a convenient way for retrofitting existing approaches and countermeasures, as well as creating new ones.

We experimentally evaluate StyxJS and find that it does not disrupt the user experience, while incurring negligible overhead in terms of page load time in the vast majority of websites. Moreover, we retrofit three existing systems as StyxJS plugins, one from the browser- and two from the JS-instrumentation family, which suffer from at least one of the aforementioned limitations (SugarCoat [28], LeakInspector [29], and ScriptProtect [12]). We experimentally demonstrate that the StyxJS-adaptations significantly outperform the original systems in their respective goals, while addressing their inherent limitations. Finally, we leverage our system to analyze 3P scripts reading *first-party* cookies, which can lead to severe attacks like session hijacking, and find that it is a widespread phenomenon.

Overall, the concept of an *origin* is, arguably, the cornerstone of any notion of security on the web. While the ability to embed scripts from other origins is vital for the operation of the modern web ecosystem, it has also blurred the boundaries between origins. Consequently, effectively disambiguating between 1P and 3P scripts, which is a *fundamental* prerequisite for countless security and privacy frameworks and countermeasures, has become exceedingly challenging. As such, we believe that StyxJS can provide the critical underpinning they require and, thus, have released it as an open source project [30].

In summary, we make the following contributions:

- We develop and open source StyxJS, a JS-layer system that achieves *robust* and *real-time* 3P script attribution.
- We provide an extensive evaluation of StyxJS, and show that it effectively captures a multitude of script inclusion methods, while preventing common and custom bypass attempts. It also does not incur page breakage and induces a negligible performance overhead in most cases.
- We analyze three existing systems and provide multiple *novel* bypass techniques, highlighting the practical gap StyxJS aims to fill. We retrofit these systems on top of StyxJS, demonstrating the significant improvements it offers, and its flexibility to accommodate different pipelines.
- We perform a novel measurement on the risk posed by third-party scripts accessing first-party cookies in the wild.

2. Motivation

One of the core challenges of conducting a study or developing a countermeasure that revolves around JS execution, is correctly and robustly disambiguating 3P from 1P code. Systems leveraging JS instrumentation [12], [19], [31] must rely on *naïve* stack walking to disambiguate 3P code, i.e., using the `Error.stack` object [32]. This

approach is known to be susceptible to evasion [10], [21], as 3P scripts can *dynamically* inject new code, mask their identity, and evade attribution. For instance, as shown in Listing 1, a 3P script makes a call to `setTimeout` with a string argument, and the newly evaluated code outputs the current stack trace. As can be seen, the *original* 3P script is not included in the stack trace, thus preventing attribution and allowing the 3P script to conceal its malicious activities. Prior browser-instrumented systems [10], [11], [23], [28], [33] have rich, low-level information on seemingly every aspect of a page’s life cycle (e.g., DOM interactions) that can result in effective 3P code attribution. Unfortunately, such systems are tied to a single, modified browser, and possibly a single version, having no immediate impact on end users. Also, in §5.1 we demonstrate that low-level instrumentation is *not* self-sufficient for robust attribution, as faulty design choices can still lead to evasion and complete defense bypasses. Moreover, certain systems of both categories need a non-negligible amount of “offline” pre-processing to operate correctly [12], [28]. For instance, SugarCoat [28] requires a researcher to first *sufficiently* browse a website to generate replacements for intrusive 3P scripts, which, by design, can only cover executed code paths. To make matters worse, this pre-processing must be run for *each* new or updated script, before its replacement is adopted by existing content blocking tools. Thus, it becomes clear that such systems’ real-world adoption and benefits for end users are significantly limited.

```
1  /* https://3p.com/evade.js */
2  setTimeout("console.log(new Error().stack);")
3  /* Output */
4  Error at <anonymous>:1:13
```

Listing 1: The dynamically evaluated code via `setTimeout` cannot be attributed to the injecting script with `Error.stack`.

Motivated by the aforementioned limitations, we propose *StyxJS*, a JS-based solution which, at its core, aims to provide real-time and robust 3P script attribution. Realized as a browser extension, our system runs *without* any *a priori* knowledge or processing of the visited websites in *real time* and covers any executed code path. Due to its plugin-based architecture StyxJS can be used to *properly* replicate certain prior approaches, as well as facilitate new pipelines, as we extensively demonstrate through real-world use cases (§5).

StyxJS targets both security and privacy practitioners, who can leverage it as the basis for new frameworks, as well as any researchers measuring and analyzing the web ecosystem. Additionally, StyxJS can aid developers in profiling and analyzing the 3P scripts embedded in their sites. Most importantly, StyxJS can be promptly adopted by end users for employing the privacy enhancements we discuss in §5, or other defenses developed in the future.

2.1. Threat Model

In the context of our work, we consider malicious 3P scripts that are loaded in a 1P website and attempt to undermine user privacy or website security, e.g., by reading sensitive information or introducing vulnerabilities. These scripts can deliberately inject dynamic scripts, have full access to *all* built-in JavaScript APIs, and can

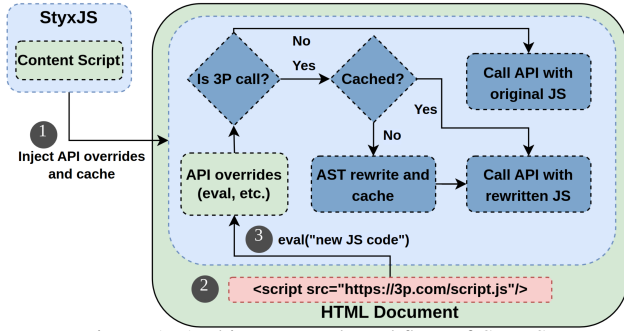


Figure 1: Architecture and workflow of StyxJS.

collude with each other or even leverage functionality offered by other benign 3P or 1P scripts. Finally, we assume that deployed standardized defenses that dictate script execution capabilities (e.g., CSP [34]) cannot be bypassed (i.e., we assume that browsers have implemented these mechanisms correctly).

3. Design and Implementation

StyxJS is comprised of different components for robustly capturing *all* 3P scripts in a page, regardless of *how* they were embedded. Before diving into the details of our approach, we provide a brief overview of StyxJS’ architecture and operation to clearly outline each component’s goals. A high-level depiction of StyxJS can be seen in Figure 1.

Content script. Our extension’s content script is injected in *all* documents and is configured to run *before* any other code in the page. This design choice allows us to inject our page scripts (which we describe next) in a timely manner, before malicious or privacy-intrusive scripts can execute. We provide more details on how we tamperproof our system against evasive scripts in §3.3. The content script’s main goal is to inject our attribution logic into the page itself, while also maintaining a cache of 3P scripts that were processed during previous visits to the same domain, so as to avoid redundant operations ①.

Page scripts. These scripts implement StyxJS’ robust and real-time attribution capabilities. In more detail, a series of API overrides is set up, so as to capture *dynamically* injected 3P scripts, such as `eval` or setting a script’s contents. Each script is then rewritten to include our attribution code, as we detail in §3.2 ③. Moreover, StyxJS adopts a plugin-based architecture, allowing for custom JS rewrites and API overrides. This, coupled with our robust attribution capabilities, constitutes a major advancement, as it enables *fine-grained* and *accurate* 3P code instrumentation, which is crucial for a wide range of past and future research efforts focusing on 3P scripts. We show how to leverage this capability to retrofit existing systems in §5, further highlighting StyxJS’ impact.

3.1. Capturing 3P Scripts

Prior to performing attribution, we need to ensure that we capture *all* 3P scripts loaded in a page. Next we define our notion of 3P scripts, and detail possible script inclusion methods and how they are handled by StyxJS.

3P scripts. Strictly speaking, a 3P script is any remote JS file that does not share the same origin [35] with

the 1P page. However, websites often leverage servers with different origins under *their* ownership, for loading resources (e.g., CDNs). For instance, Google services load scripts from *gstatic.com*, a resource server belonging to Google. Since the 1P is in control of such servers, it essentially makes their resources 1P and should be distinguished from *real* third parties. To that end, we leverage EFF’s PrivacyBadger’s list of *same-entity* domains, which is designed and maintained for this exact purpose [36].

Standard inclusion. A page can initially load a 3P script by embedding a `script` element with its `src` attribute set to the script’s remote URL; this will cause the browser to send a HTTP request, fetch the script and evaluate it ②. For the remainder of this paper, we will refer to such scripts as *top-level* scripts. Attributing API calls back to such scripts is rather trivial and does not require any special handling. Specifically, we rely on JS’ `Error` API to acquire a stack trace and inspect the script URL in each stack frame, if any. This approach was leveraged by prior work as the *sole* means for script attribution [12], [20], [29]. Unfortunately, while it is adequate for top-level scripts, it cannot robustly attribute *dynamically* injected scripts, rendering prior approaches susceptible to evasion [10], [21].

```

1 let original = setTimeout;
2 setTimeout = function() {
3   let js = arguments[0];
4   if (is_3P_call()) { // Only for 3P calls
5     js = isCached(js);
6     if (!js) js = rewriteJS(arguments[0]);
7   } return original(js);
8 }

```

Listing 2: Simplified override for `setTimeout`.

Runtime inclusion. Scripts can introduce additional 3P code *at runtime* by utilizing APIs and DOM properties, e.g., adding a text node as a child on a `script`, or calling `eval` ③. For brevity, in the remainder of this work we will overload the term API to describe all such inclusion methods, and refer to such scripts as *dynamic* scripts. We refer the reader to Table 3 (Appendix B) for a complete list of JS inclusion APIs. We used the list provided by [12] as the starting point, but extended it to include overlooked APIs. The additional APIs were identified based on our domain expertise and through experimental exploration, and documentation analysis. Overall, our system handles 34 APIs from 12 different interfaces. In order to capture 3P scripts injected via these APIs, we leverage the page scripts injected by our content script which runs on every new document *before* any other code (§3.3). Specifically, the page scripts override each of these APIs, ensuring that whenever a 3P script calls them, our override code will execute before the original.

We provide an example API override in Listing 2, which captures dynamic scripts evaluated via `setTimeout`. Initially, to disambiguate whether the call was initiated by the 1P or a *top-level* 3P script, we simply inspect the `Error` stack trace (line 4), as detailed previously. We detail how dynamic 3P scripts are attributed in the following paragraphs. If the call was initiated by 1P code, we directly call the original API with the provided argument (line 7). For 3P calls, each override’s operation depends on the API’s nature. For APIs that expect raw JS code, e.g., `eval`, we directly rewrite the

provided argument before calling the original (line 6). For markup APIs, e.g., `document.write`, we parse the given markup, locate scripts, inline event handlers and `javascript: URLs`, and rewrite these specific parts. Next, each rewritten piece of code (or markup) is cached in our extension’s storage space, keyed with the original source’s SHA-256 hash. Then, in subsequent visits to the same domain, each override computes the given argument’s hash value and queries the cache so as to avoid redundant rewrites (line 5).

Finally, we note that there exists a single JS inclusion method that is pertinent to StyxJS and cannot be handled with regular overrides, namely assigning `javascript: URLs` to `window’s location` property. This occurs because this specific property is non-configurable, thus preventing us from overriding and intercepting it. Due to space constraints, we detail the novel technique we devised to tackle even this demanding case in Appendix B.1.

3.2. AST rewriting

Having gained control over when a 3P script is dynamically evaluated in the page, we then need to rewrite each one so as to add our attribution code, while maintaining its original functionality. We provide an example dynamic script injected via `document.write`, and its rewritten version, in Listings 3 and 4 respectively. The script declares and calls an `async` function, which awaits a *promise* and evaluates a new script. To achieve our goal, we utilize the *acorn* [37], *estraparse* [38] and *aststring* [39] libraries and perform the following steps. First, we generate the source code’s AST to perform fine-grained rewrites. Next, we wrap the *entire* script, and each function body, in a `try/finally` (lines 2-10, 5-8), where we prepend our entry code before the `try` (lines 1, 4) and our exit code in the `finally` clause (lines 8,10). This guarantees that when the script or one of the functions exit, either gracefully or due to an exception, our exit code will *always* execute before anything else. Finally, we generate the final source code and return it.

```
1  async func foo(){
2    await ResolveAfter5Sec();
3    eval("alert('Dynamically injected!');");
4  } foo();
```

Listing 3: Example dynamic 3P script.

```
1  let gx = S.push("doc.write", ID);
2  try{
3    async func foo(){
4      let fx = S.push("doc.write", ID, "foo");
5      try{
6        await xxawait(fx, ResolveAfter5Sec());
7        eval("alert('Dynamically injected!');");
8      }finally{ S.pop(fx); }
9    } foo();
10 }finally{ S.pop(gx); }
```

Listing 4: Example dynamic 3P script after StyxJS’ rewrites.

Attribution code. In order to attribute dynamic 3P scripts at runtime, we need a real-time stack trace, akin to the *Error* approach for top-level scripts. In other words, we need a 3P script or function to push onto a stack when executing, and pop from that stack when exiting. To that end, we also create a globally accessible stack

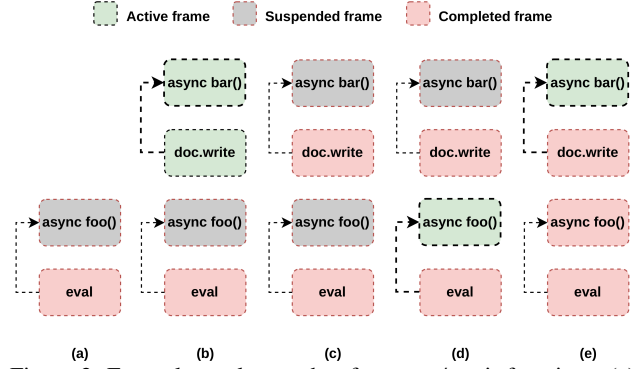


Figure 2: Example stack snapshot for `async/await` functions. (a) An `eval` script (completed) has previously called `foo`, which is now suspended due to `await`. (b) Later, a script injected with `document.write` executes and calls the asynchronous function `bar`. (c) `bar` is also suspended and the dynamic script exits. (d) `foo` resumes execution and exits. (e) `bar` resumes execution, exits and all completed frames are removed.

interface; we detail how we conceal and protect our stack from malicious 3P scripts in §3.3. As shown in Listing 4, when a script executes, our entry code will push a new frame in our 3P stack, including the API that injected the script and a unique script ID, *and* store its stack frame’s index (line 1). The same applies for functions, which also store their name or a unique identifier for anonymous functions (line 4). When a new frame is pushed, we mark it as the currently active frame *and* store the previous active frame, if any, as its parent. Essentially, our stack can only include a *single* active frame, indicating that a specific dynamic 3P script or function is executing and allowing us to correctly attribute further script injections (or various JS API calls, as discussed in §5). The exit code (lines 8,10), which aims to remove the frame, operates differently from a regular `pop` operation to account for asynchronous (*and* generator) function calls. If the exiting frame is not *last* but is followed by more recent ones, we can deduce that it must be, or was followed by, an asynchronous function that had suspended its operation until a promise was resolved. In the meantime, other dynamic 3P scripts or functions might have been executed and pushed as new frames in the stack. In such cases, we cannot simply remove the frame, as doing so would invalidate the stack indices stored by the more recent ones. Instead, we mark the frame as *completed* and return, as depicted in Figure 2. When, however, an exiting frame *is* the last active call, it simply pops itself from the stack *and* removes any previous, consecutive *completed* calls, so as to discard unneeded frames. Finally, upon exiting, we also iterate the chain of parent frames to find the currently active one; if none are found our active stack frame is set to `null`, indicating that *dynamic* 3P execution has halted.

Async await calls. Another challenge is the use of the `await` operator, which suspends the execution of the *current function* until its operand is resolved. If in the meantime the 1P calls one of the overridden APIs, our 3P stack would be non-empty and we would wrongfully attribute 1P to 3P code. To tackle this, during rewriting we wrap each `await` operand as an argument into a custom function (line 6). This function marks the current frame as *suspended*, wraps the awaited promise in a new promise and returns it to maintain functionality (Figure 2

(a) and (c)). When the original promise resolves, the wrapper promise will also resolve, propagate the results to the awaiting function, and mark the frame as *resumed* (Figure 2 (d) and (e)). As such, even if the 1P calls an API, the 3P function is marked as suspended and the call is *not* attributed to the 3P. We note that while static, this method is resilient against obfuscation, as `await` is a reserved keyword and cannot be concealed.

Generators. A challenge similar to the `await` issue stems from *generator* functions, which return an *iterator*. Such functions can use the reserved `yield` keyword to gradually return values when the iterator’s next method is called, and then suspend their operation until next is called again. To tackle this, we wrap each `yield`’s expression in a custom function that marks the stack frame as *suspended* right before yielding the value. We also wrap the entire `yield` statement in a `try/finally` and mark the frame as *resumed* in the `finally` clause, as it is the first statement that executes when resuming the function. We also note that this and the `await` issue have not been accounted for by prior work [28], [31].

Nested dynamic code injection. As can be seen in Listing 4 (line 7), the call to `eval` has *not* been rewritten. This is due to the fact that such *nested* dynamic code injections will be handled right before being evaluated, as described in the previous paragraphs. Specifically, the corresponding API override will check if our stack has an active frame, i.e., not *completed* or *suspended*, indicating that the call was initiated by a *dynamic* 3P script, and will proceed with rewriting the new script. We also construct a *JS Inclusion Graph* (JIG), which details the relationships between 3P scripts in the page, suitable both for offline and runtime analysis. In more detail, each graph node corresponds to a top-level or dynamic 3P script and is annotated with that script’s inclusion method, a unique ID and its parent’s ID so as to maintain script inclusion relationships. While the JIG is not essential for StyxJS’ operation, it can be a useful feature for various measurements analyzing 3P script dependencies.

Dynamic naming. We note that all pieces of code injected by StyxJS in these examples, e.g., stack interface and indices, have been named statically for the sake of simplicity. In reality, as we outline next, they are dynamically named so they cannot be guessed and tampered with by evasive 3P scripts.

3.3. StyxJS Concealment

To guarantee robust attribution, it is crucial to ensure that StyxJS cannot be bypassed by malicious 3P scripts. To achieve this, we employ existing configuration and design choices [12], [28], [40], as well as novel techniques tailored to our system’s architecture and *modus operandi*.

Basic protection. First, we need to ensure that our content script runs *before* anything else in a document and promptly injects our attribution logic and overrides. To that end, we set the `run_at` field in the extension’s manifest file to `document_start`. To prevent `iframe`-related attacks, e.g., retrieving original APIs from a sub-frame, we also set `all_frames` and `match_about_blank` to `true`. Furthermore, to restrict access to the original APIs and StyxJS’ variables, we utilize private blocks and block-scoped variables. To prevent malicious scripts from

tampering with StyxJS’ operations by overriding built-in APIs, we store and use references to the *original* APIs, e.g., `Array.push`. Finally, to avoid interference with other installed extensions and ensure StyxJS executes first, we leverage the approach by Picazo-Sanchez et al. [41]. Specifically, we disable and then re-enable all other extensions, so as to appear as they were installed *after* StyxJS, and execute as such. We note that our API overrides *do not* disrupt other extensions’ functionalities, as they only consider calls initiated by 3P scripts. Finally, since our overrides target prototypes and not specific objects, using `Object.freeze` [42] to disable any further mutations to them is not necessary in our case. Specifically, even if a malicious script deletes an override, it would *not* revert back to its original form. Moreover, our approach allows for further, legitimate overrides by the page *on top* of our own, ensuring compatibility. For instance, a page might override the `innerHTML` property setter so as to perform sanitization checks on the newly added markup, and *then* call StyxJS’ override.

UXID token. Our design requires the concealment of several operations, which is achieved through the use of a secret token. UXID is generated by the content script for every document, and is sufficiently long (128 bits) to resist brute-force attacks. If a 3P script were to learn its value, it would be able to circumvent StyxJS and avoid attribution. The content script sets each injected page script’s `id` attribute to UXID’s value so it can learn it too. After executing, each script removes itself from the DOM, so as not to leave any traces of its execution or UXID. As detailed next, UXID is used to conceal (i) a global object holding all StyxJS-specific functionality, (ii) the injected variables for stack frame indices in rewritten 3P scripts (see Listing 4), and (iii) the auxiliary libraries required by our system.

Dynamic variable naming. StyxJS exposes certain functions and variables that must be globally accessible by all overrides and the rewritten scripts (e.g., our stack and the original APIs). Naming these statically would enable trivial bypasses, as a script could access the original APIs or pollute our stack. As such, we bundle all functionalities under an object named after the secret UXID, making it unguessable by malicious 3P code, and set it as a non-enumerable property of the `window` object. Moreover, since `getOwnPropertyNames` and `Reflect.ownKeys` also return non-enumerable properties, we override them and strip our object from the returned list. Finally, we perform the same steps for the libraries our extension uses.

Web Accessible Resources. All page scripts and auxiliary libraries are listed as *web accessible resources* (WARs) [43] in our extension, making them accessible by *any* origin. As such, a script could probe for them and detect StyxJS [44], [45]; however, it would not learn UXID as it is dynamically generated. To prevent this, we leverage the fact that our content script executes before any other code and we know precisely which WARs it will request. As such, we utilize the `webRequest` API [46] in our extension’s background script and monitor the WARs requested by each document. If a WAR is requested more than once by the same document, we can deduce the request originates from a script probing for StyxJS’ presence and block it.

Detecting overrides. API overrides can be detected by calling the `toString` method on them and identifying discrepancies [47]. To tackle this, we store the original string value for each overridden API and override `Function.prototype.toString` to return the expected result whenever it is called on one of the APIs. We also note that this approach covers `toString` itself; even if it is recursively called on itself, our override code will never be revealed.

DOM exposing properties. Another critical aspect that could lead to circumvention of StyxJS, are the inherent DOM modifications that occur when rewriting dynamic scripts. For instance, if a 3P script sets a script’s `text` property, our override for that API will rewrite the given JS code. The 3P script could then access the rewritten property and learn UXID. Additionally, a property’s effects might be visible in other properties too, e.g., `innerHTML` will also reflect changes in `text`. Similarly, a rewritten node’s ancestors will also reflect that node’s real contents through such properties; we refer to all such properties as *exposing properties* (EPs) and detail them in Table 3. To overcome this, we initially store the *original* value of all dynamic scripts, right before rewriting them. Then, after rewriting, we add two randomly-generated markers as comments at the beginning and end of the modified source code. Next, we override all EPs’ getter methods and internally retrieve the real, modified value when called. Finally, we replace each piece of code encapsulated in our markers with its original value before returning it. We adopt the same approach for 3P functions, to prevent dynamic scripts from trivially reading their own (or each other’s) rewritten functions and learning UXID.

Frame polling. A malicious script can continuously poll for the creation of iframes, so as to exploit a race condition between when a frame is created and when our overrides are set, thus accessing the original, unprotected APIs. This can be achieved with a direct or recursive call to `setInterval` or `setTimeout`, respectively, with a zero delay. To prevent this, we block accesses to an iframe’s `contentWindow` and `contentDocument` until it has fully loaded [40]. However, scripts can also access a frame by using `window[z]`, where `z` is an index for each frame. Unfortunately, these indexed properties cannot be overridden to prevent access. To tackle this, whenever these functions are called, we wrap their callback in a function that inspects whether StyxJS has *not* been set up in any frame. In such cases, before calling the original callback, it sets *all* protected APIs in the uninitialized frame as `undefined` to prevent evasive scripts from accessing them. It also stores the original APIs in a UXID-named variable in the frame itself, so StyxJS can restore and override them when being set up. To the best of our knowledge, this bypass vector has not been accounted for by prior work.

Fingerprinting prevention. Recent work [48] has demonstrated that extensions can be uniquely fingerprinted via their DOM modifications, even if they try to conceal them, e.g., by reverting them. This is achieved by leveraging the `MutationObserver` API [49], which records and inspects *all* DOM modifications with a given callback function. StyxJS’ main operation is *not* affected by this approach, as we inject our logic *before* anything else in the page, i.e., before an observer can be setup. In

addition, all API overrides call their original counterpart, thus not performing any StyxJS-specific modifications. However, as we detail next (§3.4), there are a few cases where we need to set additional script attributes, so as to properly handle a page’s deployed security mechanisms. To prevent StyxJS from being fingerprinted due to this behavior, we also override `MutationObserver` and wrap the page provided callback in a filtering function. This function removes mutations caused by StyxJS and calls the original callback. We also note that StyxJS is not affected by other extension fingerprinting techniques [50], [51], as it does not inject any stylesheets nor does it react to user actions. Moreover, our system is also secure against the attacks by [52], where malicious scripts escalate their privileges to the extension level via message-passing, as we do not exchange any messages. Finally, we note that detection through side-channels (e.g., timing-based) are out of scope for StyxJS, which is in line with prior work [28], [40], [47], [53].

3.4. Maintaining Security Mechanisms

Agarwal et al. [54] recently found that several extensions carelessly modify page-deployed security mechanisms, diminishing their effects. In contrast, one of StyxJS’ main requirements is to maintain the security guarantees offered by deployed mechanisms and *never* introduce new attack vectors, while still being able to operate as intended. As prior work has shown, composing such security policies is often challenging due to their complexity [55], [56] or third-party restrictions [57], resulting in developers’ struggling with correct deployment [58] and subtle pitfalls that can even enable complete bypasses [59]. As such, properly *extending* these mechanisms to accommodate our system’s operation also faces these challenges and requires careful design choices, so as to not overly relax them or completely break them. Here, we detail how we handle the two mechanisms pertinent to our system’s operation.

Content Security Policy. One of the most prominent security mechanisms is the *Content Security Policy* (CSP) [34], which defines what resources and from which origins they can be loaded in the page. For instance, it could dictate that scripts can only be loaded from specific domains or if their source matches a set of predefined hash values. As such, if not handled correctly, such policies can hinder StyxJS’ operation, as our rewritten scripts would not match any allowed hash value. On the other hand, overly relaxing a CSP would diminish its security benefits and potentially open up new attack vectors. Before detailing our approach, we must note that we can only handle CSP headers upon receiving them, *before* the document is created, i.e., we do not have any knowledge of the 3P scripts that will be loaded in the page. Thus, whenever a main- or sub-frame request is intercepted, our background script detects any CSP headers and checks whether they set any directives pertinent to StyxJS’ execution. Specifically, the following three directives regulate script execution; `script-src` [60] considers all JS execution in the page and serves as a fallback for the other two. `script-src-elem` [61] considers inline scripts, while `script-src-attr` [62] regards inline event handlers, e.g., `onclick`. Finally, `default-src` [63] serves as

a generic fallback for all previous directives. If the CSP does not enforce any relevant restrictions, by not setting any of these directives, we do not need to perform any special handling.

Inline scripts. The first mechanism that regulates script execution are CSP-defined nonce values: an inline script *not* accompanied by a legal nonce will be blocked. In such cases, we do not need to perform any handling, since our operations do not affect nonces – it is up to the including script to assign scripts it injects with a valid nonce. Another way to restrict inline scripts, is defining specific hash values; if a script’s contents’ hash does not match any of them, it will be blocked. As such, StyxJS’ rewritten scripts would not match any hash value and would always be blocked. To tackle this, we initially collect all hash values from the CSP and also extend it with a unique, random nonce [64]. Then, whenever a new inline script is about to be injected, we compare its hash with the legal hashes. If it matches, indicating that the script should be executed, we add our custom nonce to the script and proceed with rewriting it. This way, CSP’s hash check will fail due to our rewrites, but the script will still execute due to our nonce addition. If the script does not match any of the given hashes, we cannot be certain that it should be blocked, e.g., the including script might add a legal nonce to the new script *after* setting its contents. In such cases, we still rewrite the script, but do *not* add our nonce; if the script turns out to be allowed due to a nonce, it will still execute, otherwise it will be blocked as would happen without StyxJS.

Inline event handlers. Finally, the `script-src-attr` and `script-src` directives can specify inline event handlers’ hash values with the `‘unsafe-hashes’` keyword. The latter also regulates the execution of `javascript: URLs`. Similar to the inline scripts, when an event handler is set on a node, e.g., via `innerHTML`, we perform the hash check internally. If the handler is allowed, we cannot extend it with our nonce, as it is not applicable in this case. Instead, we rewrite it and set it via `addEventListener`, which is *not* bound to CSP restrictions [62]. If it does not match, we rewrite it, but set it via `setAttribute`, so as to maintain CSP enforcement. Setting `javascript: URLs` in anchors’ `href` attribute, essentially behaves as a `click` event. As such, we perform the same procedure, but instead of `href`, we set its `onclick` handler. Finally, for `javascript: URLs` in `iframes’ src` attribute, we perform the hash check and, if allowed, we set the frame’s `srcdoc` (which has precedence over `src` [65]) to an equivalent markup including the rewritten JS as an inline script, carrying our custom nonce. We note that `srcdoc` iframes inherit their parent document’s CSP [66], and thus, our nonce allows the script to execute.

Trusted Types. Another mechanism related to script execution are *Trusted Types* (TT) [67], which aim to prevent DOM-XSS vulnerabilities by restricting dangerous APIs to *only* accept trusted input. Specifically, a website can set two additional CSP directives: `require-trusted-types-for` to enable TTs, and `trusted-types` to define TT policy (TTP) names to be used within the page. These TTPs are then defined in the page with the appropriate API, and can be used to create

trusted inputs, i.e., by sanitizing their string argument and converting it to a TT [68]–[70]. If malicious code was to be injected in a DOM sink, it would either be sanitized or it would not be a TT; in any case, its execution would be blocked. To enable seamless StyxJS integration, we extend the `trusted-types` directive with a randomly generated TTP name and also define it in the page. This TTP converts the given string to the appropriate TT without *any* modifications. Then, whenever a TT is provided in one of our overrides, StyxJS will first convert it to a string to perform our rewrites, and then convert it back to its initial TT using our custom no-op TTP, maintaining functionality. If the provided argument is *not* a TT but a regular string, we perform no conversions, maintaining the mechanism’s security guarantees.

CSP concealment. We also account for scripts attempting to detect StyxJS by acquiring the modified CSP and inspecting for our custom nonce and TTP. Specifically, a script can achieve this by registering a listener for `securitypolicyviolation` events [71], intentionally triggering a CSP violation and inspecting the event’s `originalPolicy` property. As such, we override the property’s getter method and strip our modifications, effectively returning the original CSP. Finally, we perform the same procedure for script elements’ `nonce` attribute.

CSP via `<meta>`. While StyxJS is designed to carefully handle CSPs, it cannot handle CSPs deployed via `meta` HTML elements. This is because browsers will directly enforce the CSP, and there is no foolproof way to modify it beforehand. Also, injecting additional `meta` CSPs can only make the initial policy *stricter* [72], while we aim to maintain the same level of security. This does not significantly impact StyxJS, as prior work has shown that such CSPs are rare [55]. In addition, such CSPs can affect StyxJS *only* if they specify script hashes or the TT directive, since these are the only cases for which we need to extend the CSP with our custom nonce or TTP. Finally, even if we do not handle such a CSP, it would only cause breakage, but would not hurt attribution, i.e., scripts would be correctly captured by StyxJS, but they would be blocked by the browser.

3.5. Manifest Version and Porting

While Chrome has rolled out Manifest V3 (MV3) [73] and aims to force migration for all extensions [74], we opted to implement StyxJS leveraging the well-established MV2. In the following paragraphs, we detail MV3’s shortcomings that led to this design choice, and detail the specific StyxJS operations that need to be ported once the relevant MV3 issues are solved by Chrome.

Script injection. When developing our system, through empirical analysis we found that MV3 did not provide a foolproof way to inject scripts *before* any other code executes in the page, thus enabling potential StyxJS-evasion vectors. Specifically, a key change is that MV3 does not allow fetching and executing remote code. While our page scripts are listed as Web Accessible Resources (WARs) [43] and reside *locally* in the extension’s directory, this change prevents us from fetching and adding them as inline scripts in the page. Adding them as `script` elements with their `src` attribute set to their local URL is allowed, but this causes Chrome to

execute them *asynchronously* and possibly *after* 3P code has executed. While the *userScripts* API [75] solves this issue, we found that it does not inject scripts in *blank* iframes. This also allows for bypasses, as evasive scripts can acquire references to unprotected APIs through such frames. In contrast, MV2 allows for the prompt injection of StyxJS’ page scripts.

WAR probing. As detailed in §3.3, we devised a novel mechanism to prevent malicious scripts from probing StyxJS’ web accessible resources and detecting its presence. This approach would not be required in our system’s MV3 version, as MV3 provides the `use_dynamic_url` manifest field [76] for preventing scripts from knowing and requesting WAR URLs.

CSP handling. We note that when modifications on deployed CSPs are needed (detailed in §3.4), they are performed on the fly using the `webRequest` API’s blocking mode, which is no longer supported in MV3. In contrast, MV3’s restrictive `declarativeNetRequest` (DNR) API [77] relies on predefined, coarse-grained rules to modify HTTP headers, and does not currently support their conditional modification. Specifically, it only allows to strip, replace or add a new header, without providing more fine-grained control over the header’s original contents. Nonetheless, this missing feature has been requested by multiple extension developers, and plans for its integration have been announced [78]–[80]. Once this is done, performing our CSP modifications in MV3 will only require writing a simple regex-based rule that checks for the existence of legal script hashes or the TT directive, and extending each one with our nonce and TTP, respectively. Alternatively, we have experimentally verified another technique for achieving this. In essence, we can use the *non-blocking* `webRequest` API (available in MV3) to collect CSP header values and then completely strip them using the DNR API. Then, our content script can inject the modified CSPs as a `<meta>` element in the page, ensuring that all policies are correctly deployed before any code executes.

Summary. Overall, when taking into consideration the aforementioned MV3 limitations and since our main goal is *robustness*, we decided to develop StyxJS’ prototype leveraging the more mature and flexible MV2. Nonetheless, we stress that our system has been designed with MV3 requirements in mind, and will be readily portable once the aforementioned issues are resolved. Finally, we note that several browsers will maintain MV2 support [81], [82], including Chrome via a minor user tweak [83].

3.6. Summary

It is apparent that achieving *correct* and *robust* JS attribution is not a simple task, as a multitude of factors must be taken into account. Apart from the straightforward task of hooking relevant APIs for dynamic JS inclusions, one needs to effectively tackle several idiosyncrasies of the language itself, such as the `await` and `yield` keywords and their effects. Similarly, dynamic 3P scripts’ and functions’ execution must be precisely monitored even in cases of early termination due to unhandled exceptions. Crucially, robustness further mandates careful design choices for tamperproofing the attribution scheme against evasive

TABLE 1: Bypass techniques tested against StyxJS.

#	Bypass	Prevention
1	3P function <code>toString</code>	Uses override to remove rewrites
2	Access original APIs via iframe	StyxJS runs on all frames on <code>document_start</code> & blocks access until fully loaded
3	Enumerate window	Secret, non-enumerable properties
4	Injected scripts’ sub-tree’s and ancestry chain’s EPs	Uses EPs’ getter overrides to filter out rewrites
5	Revert APIs by deleting them	Overrides prototypes and non-reversible APIs

scripts. Finally, any system aiming to enhance security should integrate seamlessly with *existing* security mechanisms deployed by websites and never diminish their own benefits.

4. Experimental Evaluation

Here we present our extensive evaluation of StyxJS. We experimentally prove the robustness and effectiveness of our system, measure the performance overhead it induces and study how its operation may effect real websites.

Setup. We ran all experiments using Chrome 122.0.6261.94, on a machine with a 16-core Intel Core i7-11700 @ 2.50GHz CPU and 62GB of RAM.

4.1. Validation

Attribution. To validate the efficacy of using StyxJS to capture and attribute 3P code, as well as to ensure that our system cannot be bypassed, we employed the following methodology. First, we created an HTML page that serves as the first party, which loads two external 3P scripts. These scripts leverage *all* JS inclusion methods detailed in Table 3 and dynamically inject new scripts in the page, each one comprising a specific test. As aforementioned, we initially used the JS inclusion APIs by [12] and extended them to capture several overlooked cases. As such, while more comprehensive, there might still be cases we have missed. Nonetheless, StyxJS’ modular design allows for the straightforward addition of missed APIs, or even new ones that will be introduced in the future. In addition, each test calls `insertAdjacentHTML`, so as to print its name *and* which top-level script injected it. We then created a plugin for StyxJS that overrides `insertAdjacentHTML` and validates whether the injected JS is correctly attributed to the appropriate top-level script, based on each test’s information.

As an initial validation experiment, we first ran the top-level 3P scripts sequentially, to allow each one to execute all of its tests in order. Next, to further stress the capabilities of StyxJS, we also setup the two 3P scripts to run their tests in an interleaved manner, by scheduling them via `setTimeout` in random intervals. Overall, this approach allowed us to verify that StyxJS passed all tests in *all* cases, indicating that it effectively captures both the top-level scripts and *all* dynamic scripts they inject, as well as all declared functions. In addition, StyxJS records the relationships between scripts and composes the JS inclusion graph; a shortened version of the JIG generated from the test page is depicted in Figure 4, Appendix B.

All JIG nodes, corresponding to 3P scripts in the page, are annotated with their specific inclusion method, their original source code and whether they were fetched from the cache or were rewritten on the fly. To account for JS that might have been injected but *not* executed (e.g., a 3P event handler that never fires), each node is marked as *executed* when the corresponding script is evaluated and its entry code runs. *Nested* scripts are also captured correctly and associated with their parent scripts.

Real-world validation. To validate StyxJS in real websites, we initially attempted to compare its coverage against PageGraph [22], but found cases where PageGraph yielded *incorrect* results, preventing us from performing a fair and correct validation; we provide more details in Appendix B.2. To that end, we randomly selected and visited 50 websites that embedded 3P scripts from our performance evaluation dataset (§4.2) and manually validated StyxJS’ attribution capabilities. Specifically, we created a plugin that inserted a few JS statements in each dynamic script’s entry point, enabling us to log the injection and the injected scripts’ details, e.g., the inclusion API, script URL and source code. Also, the entry code calls the debugger JS instruction, so as to set a breakpoint and pause script execution, allowing precise analysis. Then, whenever a dynamic script was executed and its breakpoint triggered, we manually analyzed its parent script’s source code and verified whether it truly injects the script that StyxJS captured. Finally, we repeated this process until no more dynamic 3P scripts were injected. We verified that in *all* cases StyxJS correctly attributed the script injections to their parent scripts, including more complex cases with nested scripts. For instance, a dynamic script created via `Function` injected another script through the same API or a 3P event handler set with `innerHTML` called `eval`; StyxJS correctly handled these as well.

Robustness. In order to validate our system’s resilience to bypass attempts, the scripts perform additional tests, each trying to uncover UXID or acquire a reference to an *unprotected* inclusion API. As discussed in §3.3, if UXID becomes known a malicious script can circumvent StyxJS. Similarly, if it gets a clean reference to an API it can inject further JS without being attributed. In Table 1 we detail each test and how StyxJS handles it. Specifically, the scripts attempt to read the rewritten 3P functions’ string representation, access unprotected APIs from subframes (both directly and via frame-polling), enumerate window properties to uncover our 3P stack, and inspect the exposing properties of themselves, their sub-tree and their ancestors. Finally, they delete overridden APIs to revert them back to their original form. Overall, UXID is never revealed and inclusion APIs remain protected due to the concealment techniques StyxJS employs (§3.3).

To also showcase our system’s relative robustness improvements in regards to prior approaches, we detail the JS inclusion APIs covered by StyxJS and two prior systems (one from the browser- and one from the JS-instrumentation families) in Table 3, Appendix B.1. As can be seen, out of the total 34 APIs StyxJS handles, PageGraph [53] can properly attribute 27 of them, while ScriptProtect [12] has missed 19 APIs and only partially captures another 7, leaving room for trivial bypasses. While these technical oversights can potentially be solved, in §5 and Appendices B.3, B.4, we demonstrate several

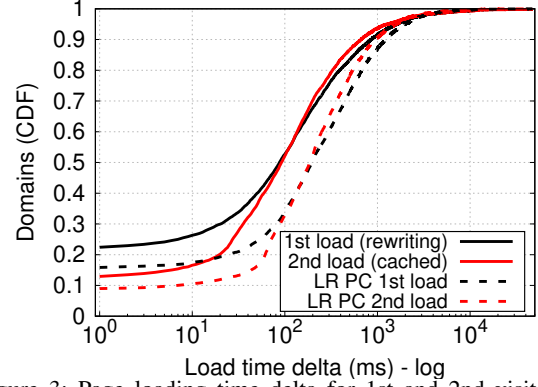


Figure 3: Page loading time delta for 1st and 2nd visit, for a modern and an older machine with limited resources (LR).

more novel bypasses against such systems that necessitate significant effort or even a complete redesign to mitigate.

Security mechanisms. To ensure that our approach described in §3.4 truly respects security mechanisms and also allows StyxJS to operate correctly, we setup our test page to deploy CSPs and TTs. For CSP we deployed a separate policy for each JS inclusion method, including inline event handlers and `javascript: URLs`, by specifying legal hash values. Specifically, we tested each inclusion method in isolation, i.e., verifying that only that method was allowed by the CSP, while all other scripts should be blocked. Similarly, we also adjusted the page to use TTs for the appropriate script inclusions. Then, for each mechanism setup, we loaded the test page both in a StyxJS-enabled and a vanilla browser, and verified that their results were consistent in all cases (i.e., legal scripts were correctly captured and executed, while illegal scripts were blocked).

4.2. Performance Evaluation

StyxJS’ complex operation imposes a performance overhead on page load times. It is crucial to measure this overhead, since StyxJS can also be deployed in end user devices. To that end, we test our system on the top 10K domains on the Tranco list [84]. Specifically, we setup a browser without StyxJS, fetch the landing page and measure its load time. Next, we refresh the page to measure the load time after some resources have been possibly cached. Then, we perform the same steps with StyxJS in place; note that StyxJS caches dynamic scripts and does not rewrite them for subsequent page loads if they have not changed. We average the load times over five runs for each browser, to account for possible network delays or other issues, and measure the overhead StyxJS induces, i.e., the corresponding delta between the browser setups for the first and second page loads.

After excluding unresponsive websites (e.g., time out, DNS errors) and those without 3P scripts, we end up with 6,522 domains, shown in Figure 3. For the first page load, where we expect a larger overhead since StyxJS needs to rewrite all dynamic 3P scripts, we find that 50% of domains require *at most* an additional 0.09 seconds to load, while 90% and 95% need up to 0.8 and 1.6 more seconds, respectively. For the second load, where our caching mechanism intervenes, half of the domains need up to an additional 0.09 seconds, 90% up to 0.65

seconds and 95% up to 1.2 seconds more to load. We observe that for half of the domains the overhead remains the same for the first and second load, i.e., our cache does not seem to provide any benefits. This is expected, as we find that 95% of scripts can be rewritten in just 0.002 seconds, a minimal amount of time that cannot be significantly reduced. Yet, for a few domains ($\sim 10\%$), our caching mechanism slightly reduces the overhead, further improving the user experience and enabling wide adoption. While a more elaborate caching mechanism which also accounts for volatile values in scripts (e.g., timestamps) could further reduce the overhead, we opted to use this more straightforward approach due to the typically negligible rewriting cost. Overall, the overhead induced by our system for the majority of the domains is negligible, as StyxJS can accurately and robustly attribute web scripts, which is crucial for the accuracy of measurement studies and the effectiveness of countermeasures (as we demonstrate in §5). Nonetheless, StyxJS supports allowlisting domains in case a user observes prohibitively larger loading times. Moreover, the storage footprint of our entire cache, spanning $\sim 6.5\text{K}$ domains and $\sim 188\text{K}$ dynamic 3P scripts, is roughly 400MB, making our approach suitable for end user deployment. Finally, these dynamic scripts were found in $\sim 5.1\text{K}$ domains (78%), demonstrating the prevalence of runtime JS inclusions.

Finally, to further assess StyxJS’ performance impact on more resource-constrained devices, we repeat the same experiment using an older commodity machine with 8GB of RAM. As shown in Figure 3, the overhead only slightly increases, with half of the domains requiring up to 0.18-0.19 and 90% up to 0.9-1.2 seconds more for the two page loads. This demonstrates that our system is suitable for deployment even when using older commodity hardware.

Breakage. To measure the breakage StyxJS might cause to real websites, we followed an approach similar to prior work [40]; while their work had a different focus, it also relied on a browser extension that heavily depended on API overriding. We randomly picked 50 out of the top 150 Tranco websites that include 3P scripts and proceeded to manually investigate them. We browsed each website *without* StyxJS, and recorded all encountered functionalities, e.g., signing up, editing settings, posting comments etc. We stress that contrary to [40], we did *not* set a time limit for this experiment, but tried to be exhaustive and uncover as many functionalities as possible. Next, having learned the website’s behavior and appearance, we revisited it with StyxJS and exercised the same functionalities. After this procedure, we verified that StyxJS did not cause any breakage, as we did not encounter any appearance or functionality issues, thus not affecting the user experience. Coupled with the low, and typically negligible overhead, this further corroborates the suitability of StyxJS for end user deployment. While our current technical implementation in MV2 may limit end-user adoption, as Chrome plans to deprecate it, we expect StyxJS to retain these properties (i.e., low overhead and limited breakage) when ported to MV3.

5. Use Cases

We designed StyxJS as a plugin-based architecture to simplify the creation of new pipelines on top of its

robust and real-time attribution capabilities. Here we pick an existing system that relies on browser instrumentation as a representative use case. Moreover, we chose another two systems relying on JS instrumentation, which we detail in Appendices B.3 and B.4 due to space constraints. We present each system alongside its limitations, and demonstrate proof-of-concept (PoC) techniques for bypassing them. Next, we retrofit them as a plugin on top of StyxJS, to highlight how our system conveniently enables a wide range of studies and systems. We also provide additional insights and discuss our improvements over the original tools. Finally, we further highlight StyxJS’s ability to analyze the web ecosystem by performing a novel measurement of 3P scripts reading 1P cookies, which can lead to severe attacks such as session hijacking.

5.1. SugarCoat

Overview. SugarCoat [28] aims to address the privacy-functionality trade-off stemming from the explicit blocking of privacy-intrusive scripts, embedded in thousands of websites, since blocking them can cause significant breakage. This is achieved through safe *resource replacements* for such scripts, which can then be adopted by content blocking tools. In more detail, SugarCoat relies on a PageGraph-enabled browser [53] being driven by a privacy researcher so as to dynamically trace privacy-related API calls by harmful 3P scripts, and then map them to their JS source code. The target scripts are rewritten so as to redirect these API calls to *mock* API implementations, thus protecting privacy while also maintaining functionality.

Limitations & bypasses. While SugarCoat addresses a significant problem, it also suffers from certain core limitations. Initially, the generation of safe replacements happens offline; they can only be adopted by existing content blocking tools at a *later* time. In other words, the system’s benefits, while practical, are not immediate for end users. In addition, this offline pipeline must be run for *each* newly seen or updated script. Finally, another core limitation, is the inherent shortcoming of its dynamic analysis approach, which *only* covers executed code paths. As such, SugarCoat’s effectiveness in generating *complete* and *safe* script replacements, heavily depends on its pre-processing’s coverage, as we outline next. Based on these limitations and our analysis, we have devised bypass techniques that a malicious 3P script can use to avoid attribution and maintain access to privacy-sensitive APIs.

Executed code paths. SugarCoat can only capture sensitive API calls that were executed during its offline pre-processing. A malicious 3P script can exploit this limitation and alter its behavior depending on whether it executes in its original form or if it is a SugarCoat-generated replacement. To illustrate this, consider a 3P script that originally has the following structure:

```

1 func oracle(){ret !cookie.includes("abc")}
2 if(oracle()){
3   func snoop(){ret storage.getItem("KEY")}
4 }else{ // Declare same function
5   func snoop(){ret storage.getItem("KEY")}
6 } let secret = snoop();

```

Listing 5: SugarCoat-evading 3P script.

The script leverages an oracle to determine if it is a SugarCoat replacement or not. In this case, the 3P author has crawled a large number of domains embedding their script, and has collected information on what cookies each one sets. As such, if the script runs in the PageGraph browser, which does not impose any API restrictions, it will be able to access this information (line 1). Then, regardless of the result, it performs the exact *same* operation, e.g., accessing a sensitive value in `localStorage` with the `snoop` function (6); the difference lies in *where* this function is declared (3, 5). When the script is analyzed, the oracle will determine it is *not* a replacement and SugarCoat will map the `localStorage` access to the `snoop` function declared in the `else` clause (5). Therefore, the script replacement will have the following structure (simplified for brevity):

```

1 mockCookie={...}; mockStorage={...};
2 func oracle(){ // Replaced with mock API
3   ret !mockCookie.includes("abc");
4 }
5 if(oracle()){
6   func snoop(){ret storage.getItem("KEY")}
7 }else{
8   func snoop(){ // Replaced with mock API
9     ret mockStorage.getItem("KEY");
10  }
11 } let secret = snoop();

```

Listing 6: SugarCoat-evading 3P script replacement.

When adopted by a content blocking tool and ran in a user’s browser, the replacement script will again call its oracle. However, the `cookie` API has been replaced with a mock implementation (3) and the oracle will not be able to retrieve the expected cookie, indicating that this is in fact a SugarCoat replacement script. As such, the `snoop` function, now declared in the `if` statement (6), will access the *original* `localStorage` API, fetching the `secret` value and effectively bypassing SugarCoat’s protection. We note that the oracle in this example is rather simplified for demonstration purposes. In practice, since the mock APIs adopted by content blocking tools are public knowledge, 3P script authors can implement much more elaborate oracles, e.g., by leveraging several APIs, looking for discrepancies that indicate if their script is in its original form or a replacement.

Stringifying functions. A more direct approach to bypass SugarCoat relies on its oversight to prevent scripts from retrieving their functions’ string representation. Specifically, SugarCoat assigns the original API to a variable, so as to restore it after the script’s execution, and dynamically names it to prevent scripts from guessing it. However, a script can call a function’s `toString`, extract the variable’s name and restore the original API right before using it. We also note that this approach can be used as a generic oracle for the first bypass technique we demonstrated, as a 3P script can trivially check if one of its functions has been rewritten by SugarCoat.

PageGraph oversights. Another limitation we identified, is that PageGraph fails to correctly capture string-to-code evaluations via `setTimeout` and `setInterval` calls, as well as several `javascript: URLs`, e.g., in anchors’ and areas’ `href` attribute. Specifically, the system adds each script as a node in the final graph and connects it to the rest of the graph with an *execute* edge,

TABLE 2: API calls captured by SugarCoat and StyxJS.

	Network	Storage + cookie	Navigator
SugarCoat	2,494	15,577	9,103
StyxJS	3,547	76,904	64,865

allowing it to attribute the injection back to another node. In contrast, the aforementioned script inclusions are not connected to any other node, preventing attribution. As a result, a 3P script can simply inject all of its functionality, including accesses to privacy sensitive APIs, through a single call to one of these functions and evade SugarCoat’s rewriting. It is important to note that this specific bypass technique is not tied to SugarCoat, but affects any system that relies on PageGraph for script provenance [11], [22], [33], [85], [86]. We stress that we experimentally verified each bypass technique, by writing PoC evading scripts that access sensitive APIs and then generating their replacements via SugarCoat. Our findings were disclosed to and acknowledged by the Brave browser [87], which issued fixes for the PageGraph-related bypasses and awarded us with a bounty.

StyxJS adaptation. Adapting SugarCoat on top of StyxJS was straightforward, as we only needed to implement the mock APIs, adjust them to utilize our 3P attribution and inject them on each page. SugarCoat’s pre-processing is not applicable in our case, as StyxJS operates in real-time and does not require *any* prior knowledge of the websites or the scripts they include.

In more detail, we created mocks for the same APIs as [28] (`storage`, `cookie`, `navigator`, `XMLHttpRequest`, and `Fetch`), but instead of *replacing* and *restoring* the APIs, we override their prototypes’ properties and functions to disambiguate between 1P and 3P calls at runtime, by utilizing our 3P attribution scheme. Then, whenever an API is called by a target privacy-intrusive 3P script, the override will redirect the call to the mock API instead of the real one. This has two significant advantages; first, we do not need to locate *where* the API is called since we rely on dynamic code interception and, thus, cover *any* executed codepath. This makes our approach robust against evasion techniques such as the one presented in Listing 5. Moreover, it solves another limitation, related to `async` calls. As [28] mentions, if a 3P `async` function accesses a protected API and is temporarily suspended, other benign code will access the mock API instead of the original, as it has not been restored yet, until the function resumes and completes. Disambiguating API calls at runtime, coupled with our `await` handling (§3.2), solves this issue as the function will be marked as suspended, enabling correct API call attribution. Overall, our plugin, including mock APIs and recording protected API calls for analysis, totals 274 lines of code (LoC), while the original mock APIs [88] need 465 LoC. This is because SugarCoat has to replicate each API in its entirety and properly mimic its behavior, due to its `replace & restore` approach. In contrast, StyxJS essentially extends the original API and can utilize existing properties. For instance, a script can create a regular `XHR` object without any intervention from StyxJS, but sending the request will be blocked by the corresponding override.

Comparison. Next, we utilized our plugin to measure its effectiveness and compare it to the original version. Initially, we collected the filter list rules generated by

SugarCoat [89], which define specific privacy-invasive scripts that should not be blocked for compatibility. Next, we collected the current script *exception* rules from the same filter lists as [28] (EasyList [90], EasyPrivacy [91], Brave [92] and uBlock Origin (uBO) [93]), so as to find domains that likely included these scripts at the time of our experiment. This procedure yielded 3,025 domains. We then set up a browser with uBO, configured to redirect target 3P scripts to the *original* SugarCoat replacements [89], [94], as intended by their system. We also modified SugarCoat’s script replacements to record API calls. Next, we visited each domain and waited for 5 seconds after the load event to allow scripts to execute. Finally, we repeated the same process for these domains with StyxJS’ plugin, targeting the *same* 3P scripts. Overall, we found 1,616 domains that were responsive and included at least one target script.

As shown in Table 2, the API calls protected by the original SugarCoat scripts are significantly fewer than the ones covered by StyxJS. To further shed light on the underlying causes of this discrepancy, we sampled and manually inspected 20 of SugarCoat’s script replacements in an attempt to identify whether relevant API call sites were missed during its offline pre-processing. Through this process, we found that 18 out of 20 script replacements actually missed *several* privacy-invasive API calls and only covered a subset of them. For instance, for the popular Google Analytics script, SugarCoat captured a few calls to `navigator.userAgent` and `XMLHttpRequest`. However, the script also reads and writes cookies several times and sends additional requests, which were not replaced by SugarCoat’s mock APIs. In another script, apart from missing API calls to `storage`, `cookie` and `navigator` properties, the script replacement threw an exception due to SugarCoat’s rewrites, preventing it from fully executing and possibly reaching further API calls. These findings further highlight two crucial SugarCoat shortcomings. First, scripts do not always expose *all* of their functionalities, making offline pre-processing inadequate. Second, SugarCoat’s script replacements were generated over three years ago, making them obsolete and error-prone. It also highlights that generating one-off script replacements is not a robust, long term solution, as one needs to *constantly* generate replacements for new or updated scripts (and push them upstream to content blocking tools). To better illustrate this limitation, we collected the scripts that current filter lists exempt from blocking and visited them daily for one week. We found that from 238 scripts that were responsive, 10% were updated once, while another 10% changed multiple times. In other words, SugarCoat would have to pre-process these scripts several times in a single week to accurately capture their current version. In contrast, StyxJS does not suffer from these limitations as it captures the *current* 3P script versions in real time, without prior processing, and covers *all* executed code paths.

When comparing our plugin’s performance overhead to the original system, we find that for the first page load it requires up to 0.16 and 1 seconds more for 50% and 90% of the domains respectively. For subsequent loads, the overhead is reduced to up to 0.12 and 0.77 seconds more for the same fraction of domains. We also performed a breakage comparison on 50 randomly chosen domains, to

test whether StyxJS causes additional breakage. We visited each website with both the original system and StyxJS’ plugin and exercised the various functionalities. We found one domain in which images would not load correctly with SugarCoat, possibly due to its script replacements being outdated. Most importantly, we encountered no issues stemming from StyxJS’ operation.

Further insights. Since SugarCoat’s rules and script replacements target *specific* scripts, without accounting for dynamic GET parameters or paths in script URLs (e.g., client IDs or version numbers), we relaxed the filter rules by replacing dynamic parts with wildcards. We also collected the scripts that are currently exempt from blocking in the various lists. We then set up StyxJS to run for all 3,025 domains, out of which 2,127 (70%) included at least one privacy-harming, but compatibility-critical script, which current filter lists allow. Overall, our plugin prevented 7,184 network related, 274,284 `storage` and `cookie` and 236,073 `navigator` API calls, *without any* a priori analysis of the scripts. This further corroborates StyxJS’ practicality and immediate privacy benefits for users, as it can promptly target such scripts solely based on their URLs.

5.2. Cookie Stealing

A significant threat posed by malicious 3P scripts is their ability to exfiltrate cookies set by 1P websites that have not been protected with the `httpOnly` attribute and are accessible by JS [95]. In fact, such *cookie stealing* attacks have severe ramifications as cookies can also expose sensitive information and even lead to session hijacking, granting attackers access to users’ accounts. Importantly, a prior study by Drakonakis et al. [13] showed that ~23% of websites are vulnerable to such JS-based cookie hijacking attacks. As such, given the prevalence of susceptible websites and the attack’s severity, we decided to measure the prevalence of 3P scripts reading 1P cookies and, thus, their potential for stealing user information or hijacking sessions en masse. To do this, we created a StyxJS plugin that overrides the `document.cookie` [96] and `CookieStore` [97] APIs and attributes read operations to the calling script. We note that `document.cookie` and certain methods of `CookieStore` return the *entire* cookie jar to the calling scripts. Finally, we also monitor which cookies are set by each script, if any.

Measurement. We utilized our plugin to revisit the landing page of the top 10K domains and waited five seconds after each page load to allow scripts to execute and set or read cookies. We found that 90% (6,054) of the responsive domains which embed 3P scripts contained at least one 3P script that read *all* cookies. In total, this behavior was observed in 20,510 different scripts (based on their URLs), 18,618 of which have unique source code hashes. By correlating the read operations with the cookies set by each script, we observed that an alarming 43% of 3P scripts that read all cookies do so *without* having set any cookies themselves. While we cannot ascertain each script’s motives, this behavior constitutes a concerning finding as a significant number of 3P scripts tends to read cookies set by other parties. Finally, we stress that our findings are a lower bound, as a recent study by Rautenstrauch et al. [98] found that websites tend to include

more 3P scripts *post* authentication, further exacerbating the potential for session hijacking attacks.

6. Discussion & Limitations

In this section we discuss limitations of our approach and a potential direction for wider adoption.

Eval. A common issue in overriding `eval` is that it is not executed in the same scope as the code that called it [12], [31], which might cause breakage. Recently, [33] showed that only 1.9% of 1P and 3P scripts use `eval`; even this small fraction is an upper bound, as not all calls will cause breakage. Indeed, during our large-scale performance evaluation (§4.2) only $\sim 8\%$ of all `eval` calls threw an exception. This is also an upper bound as we do not expect all cases to be caused by StyxJS. To further investigate the possible impact of this issue, we inspected 20 randomly-selected domains where an `eval` threw an exception, and found only one case where it caused actual website breakage, by preventing a login modal from appearing.

JS bundling. If a website bundles several JS files into one, StyxJS would be unable to attribute API calls to their true origin. The same applies if 3P scripts are hosted in the 1P domain or inlined in its HTML. This is an inherent limitation of *any* system attempting script attribution and is not tied to StyxJS. Nonetheless, recent work [99] has shown that 3P bundles are way more prevalent than 1P ones; in such cases StyxJS can capture the entire bundle, but not distinguish individual scripts. Finally, as proposed by [99], studying bundles’ API usage so as to reduce their security impact is an interesting future direction that could significantly benefit from StyxJS’ capabilities.

Web workers. We note that service workers, and web workers in general, do not need any additional handling from StyxJS. Initially, workers can only be registered *and* hosted by the first-party origin. However, 1P workers can `import` [100], [101] 3P modules and scripts at runtime. Nonetheless, since workers operate in their own isolated execution environment and do not have access to a page’s execution context and the various APIs, they would also need an accompanying *in-page*, regular 3P script to access them. In that case, that regular script would have already been handled by our system, enabling correct attribution. Finally, since `fetch` events can only be registered by the main 1P worker script, imported 3P modules *cannot* directly tamper with HTTP responses (e.g., to redirect 1P script requests to their own malicious content).

WebAssembly. WebAssembly (Wasm) is a binary instruction format that enables the compilation of various programming languages to Web-compatible executable modules, fostering performance and portability [102]. While a page can load Wasm from third-party domains, we do not need to take any additional measures to ensure correct attribution. This is due to the fact that Wasm does not have direct access to Web APIs [103], including the ones pertinent to StyxJS, and can only utilize them by calling regular (3P) JS code, which we have already captured. Moreover, even if Wasm does acquire access to native Web APIs in the future, the `Error.stack` object will convey information on whether an API call was initiated by a 3P module or not, as per the specification [104]. This would

allow StyxJS to handle 3P Wasm modules like regular top-level 3P scripts and maintain its attribution capabilities.

Browser adoption. Ideally, web script attribution could be addressed internally by browsers and exposed via a *native* JS API as a substitute for StyxJS’ capabilities. Moreover, such an addition could potentially solve our aforementioned limitations, while also being extremely useful for security and privacy practitioners, as it would facilitate the development of a plethora of different countermeasures. As such, having established the significant improvements StyxJS offers, we hope our work can incentivize browser adoption.

Potential bypasses. Similar to the various bypass techniques we devised for other systems in our work, we note that there *might* exist bypasses for StyxJS as well. This can potentially occur due to overlooked JS inclusion APIs or the deployment of new APIs in the future. In any case, the inclusion of such APIs in our system is trivial, requiring only the implementation of the respective API overrides. Moreover, an active adversary could potentially identify a loophole in StyxJS’ instrumentation logic, leading to a novel evasion vector. By open-sourcing our tool, we hope to promptly identify any such *potential* cases, and further harden its robustness.

Ethical considerations. During the course of our work, we took precautions to minimize the potential harm to real websites and users, while maximizing the benefits provided by our results. Initially, our measurements comprised of *passive* crawls to sites’ landing pages, i.e., we did not automatically post content or otherwise cause state changes that might affect real web users. This approach strikes a balance between minimizing the traffic load sent to individual websites, i.e., just a few HTTP requests, while enabling us to extract useful results and properly evaluate our system. The only exception were the various *manual* breakage experiments we performed, where we also posted sentiment-neutral content to test different functionalities. Finally, we responsibly disclosed the novel bypass techniques we devised against PageGraph and SugarCoat to the Brave browser. This led to them applying fixes to the former, thus resulting in tangible security benefits to end users.

7. Related Work

A significant body of work has studied various aspects of 3P scripts and proposed countermeasures. Here we outline the most prominent studies, categorized by their core design.

Browser instrumentation. Several systems have been proposed for restricting scripts according to specified security policies [24]–[26], [105]–[107] and manually-curated task capabilities [23]. Apart from requiring heavy browser modifications, they all require significant manual work by developers, e.g., using new APIs, or have extravagant dependencies (e.g., virtual browsers [108] or replacing the native JS interpreter [109]), limiting their practicality. In contrast, StyxJS is *transparent* to the hosting application and does not require any such interventions.

Prior work has also studied 3P scripts from the angle of user tracking and content blocking. Iqbal et al. [10] designed AdGraph, an ML approach relying on page structure, behavior and requests to detect ads and tracking

resources; they also highlighted the need for robust script attribution. The following systems were built on top of PageGraph [22], which captures further page interactions and is more robust than AdGraph. Sjösten et al. [22] trained a classifier to detect ads in a language-agnostic fashion and generate filter lists. Chen et al. [11] relied on JS execution signatures to detect scripts evading filter lists, while Smith et al. [85] trained a classifier to measure the breakage caused by filter lists. Jueckstock et al. [86] also studied website breakage, focusing on 3P storage restrictions. Finally, Sarker et al. [33] relied on concealed API usage to detect obfuscated scripts; their system leveraged a combination of PageGraph, for script provenance, and VisibleV8 [47] (VV8) for API call detection. The common thread among all these works is that PageGraph (and VV8) is a *passive* monitoring tool. In other words, while extremely useful for offline pre-processing or post-mortem analyses, it cannot be used as-is to provide real time protection mechanisms for end users, as we extensively outlined in §5.1. In contrast, as we have demonstrated through various real-world use cases, StyxJS has the ability to fill this gap and provide a convenient method for *active* script interception, facilitating the development of effective defense mechanisms.

JS instrumentation. A few systems have been proposed to sandbox JS, either holistically or for 3P scripts specifically. Phung et al. [110] designed a Firefox-specific JS reference monitor relying on prototype patching. Their system could not run in iframes and considered *all* scripts. Agten et al. [111], proposed JSand to sandbox 3P scripts through proxy objects; it is unclear how they handled objects that cannot be wrapped in proxies [47]. Moreover, both systems required developers to write specific security policies. Tran et al. [31] implemented JaTE, a Firefox extension to impose object-level access control among scripts; they also resorted to some straightforward script rewriting, but also utilized proxies and did not handle deployed CSPs to accommodate their rewritten scripts. Moreover, both JaTe and JSand had limited consideration of JS inclusion methods. Terrace et al. [112] sandboxed 3P scripts in a virtual DOM; however, 1P code needed to utilize custom methods to interact with 3P code. Several other works that performed different types of measurements [7], [19], [113]–[115] or proposed various countermeasures [15], [20], [42], [116], either relied on naive stack walking to capture 3P calls or did not even disambiguate between 1P and 3P scripts. As such, we believe several of these systems to be excellent candidates for retrofitting on top of StyxJS to enhance their capabilities.

8. Conclusions

The ability to correctly and robustly (i.e., even in the face of malicious evasive scripts) differentiate between 1P and 3P scripts is the linchpin of a multitude of security and privacy countermeasures and policy-enforcement mechanisms. However, the ability to deploy these mechanisms and guarantee their effectiveness is undermined when confronted with embedded 3P scripts, as they are executed within the boundaries of the first party’s origin. This limitation also impacts web measurement studies, which may underestimate the magnitude and effect of certain threats due to the inability to truly disambiguate 1P

and 3P code. To address that gap, we proposed StyxJS, a framework for effectively providing real-time web script attribution. Our system has been tailored to account for a multitude of dynamic script inclusion methods and to tackle JavaScript idiosyncrasies that can hinder correct attribution. At the same time, our design has been guided towards precisely preventing common and custom evasion tactics employed by malicious scripts, while also respecting page-deployed security mechanisms. We conducted an extensive experimental evaluation that highlights the effectiveness and capabilities of our system, and also demonstrated how vastly different security systems can be retrofitted on top of StyxJS. We believe that our system provides functionality that is invaluable for a wide range of security mechanisms and analysis pipelines, and we have open sourced it so that security researchers and practitioners can leverage it.

Acknowledgment

We thank the anonymous reviewers for their helpful feedback. This project was supported by the National Science Foundation (CNS-2211574, CNS-2143363). The views in this paper are only those of the authors and may not reflect those of the US Government or the NSF. This project has also received funding from the European Union’s Horizon Europe research and innovation programme under the Grant Agreement No 101120779.

References

- [1] “Get started with Google Analytics — Google Analytics for Firebase,” 2022, <https://firebase.google.com/docs/analytics/get-started?platform=web>.
- [2] “AdSense — Google for Developers,” 2023, <https://developers.google.com/adsense/>.
- [3] “Web - Facebook Login - Documentation - Facebook for Developers,” 2022, <https://developers.facebook.com/docs/facebook-login/web>.
- [4] “Integrating Google Sign-In into your web app — Google Sign-In for Websites — Google Developers,” 2022, <https://developers.google.com/identity/sign-in/web/sign-in>.
- [5] “fingerprintjs: Browser fingerprinting library.” 2024, <https://github.com/fingerprintjs/fingerprintjs>.
- [6] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, “You Are What You Include: Large-Scale Evaluation of Remote Javascript Inclusions,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.
- [7] T. Urban, M. Degeling, T. Holz, and N. Pohlmann, “Beyond the Front Page: Measuring Third Party Dynamics in the Field,” in *Proceedings of The Web Conference*, 2020.
- [8] M. Ikram, R. Masood, G. Tyson, M. A. Kaafar, N. Loizon, and R. Ensafi, “The Chain of Implicit Trust: An Analysis of the Web Third-Party Resources Loading,” in *The World Wide Web Conference*, 2019.
- [9] D. Kumar, Z. Ma, Z. Durumeric, A. Mirian, J. Mason, J. A. Halderman, and M. Bailey, “Security Challenges in an Increasingly Tangled Web,” in *Proceedings of The Web Conference*, 2017.
- [10] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq, “AdGraph: A Graph-Based Approach to Ad and Tracker Blocking,” in *IEEE Symposium on Security and Privacy*, 2020.
- [11] Q. Chen, P. Snyder, B. Livshits, and A. Kapravelos, “Detecting Filter List Evasion with Event-Loop-Turn Granularity JavaScript Signatures,” in *IEEE Symposium on Security and Privacy*, 2021.

- [12] M. Musch, M. Steffens, S. Roth, B. Stock, and M. Johns, "Script-Protect: Mitigating Unsafe Third-Party JavaScript Practices," in *Proceedings of the ACM Asia Conference on Computer and Communications Security*, 2019.
- [13] K. Drakonakis, S. Ioannidis, and J. Polakis, "The Cookie Hunter: Automated Black-Box Auditing for Web Authentication and Authorization Flaws," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2020.
- [14] S. Englehardt and A. Narayanan, "Online Tracking: A 1-Million-Site Measurement and Analysis," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- [15] U. Iqbal, S. Englehardt, and Z. Shafiq, "Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors," in *IEEE Symposium on Security and Privacy*, 2021.
- [16] J. Su and A. Kapravelos, "Automatic Discovery of Emerging Browser Fingerprinting Techniques," in *Proceedings of The Web Conference*, 2023.
- [17] Q. Chen, P. Ilia, M. Polychronakis, and A. Kapravelos, "Cookie Swap Party: Abusing First-Party Cookies for Web Tracking," in *Proceedings of the Web Conference*, 2021.
- [18] I. Sanchez-Rola, M. Dell'Amico, D. Balzarotti, P.-A. Vervier, and L. Bilge, "Journey to the Center of the Cookie Ecosystem: Unraveling Actors' Roles and Relationships," in *2021 IEEE Symposium on Security and Privacy*, 2021.
- [19] U. Iqbal, C. Wolfe, C. Nguyen, S. Englehardt, and Z. Shafiq, "Khaleesi: Breaker of Advertising and Tracking Request Chains," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.
- [20] S. Siby, U. Iqbal, S. Englehardt, Z. Shafiq, and C. Troncoso, "WebGraph: Capturing Advertising and Tracking Information Flows for Robust Blocking," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.
- [21] C. Cursinger, B. Livshits, B. Zorn, and C. Seifert, "ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection," in *20th USENIX Security Symposium (USENIX Security 11)*, 2011.
- [22] A. Sjösten, P. Snyder, A. Pastor, P. Papadopoulos, and B. Livshits, "Filter List Generation for Underserved Regions," in *Proceedings of The Web Conference*, 2020.
- [23] W. Luo, X. Ding, P. Wu, X. Zhang, Q. Shen, and Z. Wu, "ScriptChecker: To tame third-party script execution with task capabilities," in *Proceedings of the Network and Distributed System Security Symposium*, 2022.
- [24] Z. Wang, W. Meng, and M. R. Lyu, "Fine-Grained Data-Centric Content Protection Policy for Web Applications," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2023.
- [25] M. Zhang and W. Meng, "JSISOLATE: Lightweight In-browser JavaScript Isolation," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.
- [26] Y. Zhou and D. Evans, "Understanding and Monitoring Embedded Web Scripts," in *IEEE Symposium on Security and Privacy*, 2015.
- [27] C. Jackson and A. Barth, "ForceHTTPS: Protecting High-Security Web Sites from Network Attacks," in *The World Wide Web Conference*, 2008.
- [28] M. Smith, P. Snyder, B. Livshits, and D. Stefan, "SugarCoat: Programmatically Generating Privacy-Preserving, Web-Compatible Resource Replacements for Content Blocking," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2021.
- [29] A. Senol, G. Acar, M. Humbert, and F. Z. Borgesius, "Leaky Forms: a study of email and password exfiltration before form submission," in *31st USENIX Security Symposium (USENIX Security)*, 2022.
- [30] "StyxJS repository," 2025, <https://gitlab.com/kostasdrk/styxjs>.
- [31] T. Tran, R. Pelizzi, and R. C. Sekar, "JaTE: Transparent and Efficient JavaScript Confinement," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015.
- [32] "Error.prototype.stack - JavaScript — MDN," 2022, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error/Stack.
- [33] S. Sarker, J. Jueckstock, and A. Kapravelos, "Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage," in *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2020.
- [34] "Content Security Policy (CSP) - HTTP — MDN," 2023, <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>.
- [35] "Same-origin policy - Security on the web — MDN," 2023, https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.
- [36] "privacybadger/src/js/multiDomainFirstParties.js at master · EFForg/privacybadger," 2023, <https://github.com/EFForg/privacybadger/blob/master/src/js/multiDomainFirstParties.js>.
- [37] "acorn - npm," 2023, <https://www.npmjs.com/package/acorn>.
- [38] "estraverse - npm," 2021, <https://www.npmjs.com/package/estraverse>.
- [39] "aststring - npm," 2023, <https://www.npmjs.com/package/aststring>.
- [40] S. Karami, F. Kalantari, M. Zaeifi, X. J. Maso, E. Trickel, P. Ilia, Y. Shoshitaishvili, A. Doupe, and J. Polakis, "Unleash the Simulacrum: Shifting Browser Realities for Robust Extension-Fingerprinting Prevention," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.
- [41] P. Picazo-Sanchez, J. Tapiador, and G. Schneider, "After you, please: Browser Extensions Order Attacks and Countermeasures," *International Journal of Information Security*, 2020.
- [42] M. Schwarz, M. Lipp, and D. Gruss, "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks," in *Proceedings of the Network and Distributed System Security Symposium*, 2018.
- [43] "web accessible resources - Mozilla — MDN," 2023, https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/web_accessible_resources.
- [44] A. Sjösten, S. Van Acker, and A. Sabelfeld, "Discovering Browser Extensions via Web Accessible Resources," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017.
- [45] S. Karami, P. Ilia, K. Solomos, and J. Polakis, "Carnus: Exploring the Privacy Threats of Browser Extension Fingerprinting," in *27th Annual Network and Distributed System Security Symposium, NDSS*, 2020.
- [46] "webRequest - Mozilla — MDN," 2023, <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest>.
- [47] J. Jueckstock and A. Kapravelos, "VisibleV8: In-browser Monitoring of JavaScript in the Wild," in *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2019.
- [48] K. Solomos, P. Ilia, N. Nikiforakis, and J. Polakis, "Escaping the Confines of Time: Continuous Browser Extension Fingerprinting Through Ephemeral Modifications," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2022.
- [49] "MutationObserver - Web APIs — MDN," 2023, <https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver>.
- [50] P. Laperdrix, O. Starov, Q. Chen, A. Kapravelos, and N. Nikiforakis, "Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [51] K. Solomos, P. Ilia, S. Karami, N. Nikiforakis, and J. Polakis, "The Dangers of Human Touch: Fingerprinting Browser Extensions through User Actions," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.
- [52] A. Fass, D. F. Somé, M. Backes, and B. Stock, "DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2021.
- [53] "PageGraph · brave/brave-browser Wiki · GitHub," 2023, <https://github.com/brave/brave-browser/wiki/PageGraph>.

- [54] S. Agarwal, "Helping or Hindering? How Browser Extensions Undermine Security," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2022.
- [55] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, "Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies," in *Proceedings of the Network and Distributed System Security Symposium*, 2020.
- [56] J. So, M. Ferdman, and N. Nikiforakis, "The More Things Change, the More They Stay the Same: Integrity of Modern JavaScript," in *Proceedings of The Web Conference*, 2023.
- [57] M. Steffens, M. Musch, M. Johns, and B. Stock, "Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI," in *Proceedings of the Network and Distributed System Security Symposium*, 2021.
- [58] S. Roth, L. Gröber, M. Backes, K. Krombholz, and B. Stock, "12 Angry Developers - A Qualitative Study on Developers' Struggles with CSP," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2021.
- [59] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- [60] "CSP: script-src - HTTP — MDN," 2023, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src>.
- [61] "CSP: script-src-elem - HTTP — MDN," 2023, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src-elem>.
- [62] "CSP: script-src-attr - HTTP — MDN," 2023, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src-attr>.
- [63] "CSP: default-src - HTTP — MDN," 2023, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/default-src>.
- [64] "nonce - HTML: HyperText Markup Language — MDN," 2023, https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/nonce.
- [65] "4.8.2 The iframe element — HTML5," 2010, <https://www.w3.org/TR/2010/WD-html5-20100624/the-iframe-element.html>.
- [66] "Content Security Policy Level 3," 2023, <https://www.w3.org/TR/CSP3/#security-inherit-csp>.
- [67] "Trusted Types API - Web APIs — MDN," 2024, https://developer.mozilla.org/en-US/docs/Web/API/Trusted_Types_API.
- [68] "TrustedScript - Web APIs — MDN," 2023, <https://developer.mozilla.org/en-US/docs/Web/API/TrustedScript>.
- [69] "TrustedScriptURL - Web APIs — MDN," 2023, <https://developer.mozilla.org/en-US/docs/Web/API/TrustedScriptURL>.
- [70] "TrustedHTML - Web APIs — MDN," 2023, <https://developer.mozilla.org/en-US/docs/Web/API/TrustedHTML>.
- [71] "SecurityPolicyViolationEvent - Web APIs — MDN," 2023, <https://developer.mozilla.org/en-US/docs/Web/API/SecurityPolicyViolationEvent>.
- [72] "Content Security Policy Level 3," 2023, <https://w3c.github.io/webappsec-csp/#multiple-policies>.
- [73] "Overview of the Chrome Extension Manifest V3 - Chrome for Developers," 2023, <https://developer.chrome.com/docs/extensions/mv3/intro/mv3-overview/>.
- [74] "Resuming the transition to Manifest V3 — Blog — Chrome for Developers," 2023, <https://developer.chrome.com/blog/resuming-the-transition-to-mv3>.
- [75] "chrome.userScripts — API — Chrome for Developers," 2024, <https://developer.chrome.com/docs/extensions/reference/api/userScripts>.
- [76] "Manifest - Web Accessible Resources — Extensions — Chrome for Developers," 2023, <https://developer.chrome.com/docs/extensions/reference/manifest/web-accessible-resources>.
- [77] "chrome.declarativeNetRequest — API — Chrome for Developers," 2024, <https://developer.chrome.com/docs/extensions/reference/api/declarativeNetRequest>.
- [78] "Extension using v3 to modify a part of response headers value in declarativeNetRequest," 2024, <https://issues.chromium.org/issues/40794461>.
- [79] "Blocking/Redirect request based on response headers," 2024, <https://issues.chromium.org/issues/40727004>.
- [80] "[Feature request] manifest v3: blocking responses based on 'content-type' header," 2024, <https://issues.chromium.org/issues/40657203>.
- [81] "Manifest v3 in Firefox: Recap and Next Steps — Mozilla Add-ons Community Blog," 2022, <https://blog.mozilla.org/addons/2022/05/18/manifest-v3-in-firefox-recap-next-steps/>.
- [82] "Opera, Brave, Vivaldi to ignore Chrome's anti-ad-blocker changes, despite shared codebase — ZDNET," 2019, <https://www.zdnet.com/article/opera-brave-vivaldi-to-ignore-chromes-anti-ad-blocker-changes-despite-shared-codebase/>.
- [83] "Chrome Enterprise Policy List and Management — Documentation," 2024, <https://chromeenterprise.google/policies/#ExtensionManifestV2Availability>.
- [84] "A research-oriented top sites ranking hardened against manipulation - Tranco," 2023, <https://tranco-list.eu/>.
- [85] M. Smith, P. Snyder, M. Haller, B. Livshits, D. Stefan, and H. Haddadi, "Blocked or Broken? Automatically Detecting When Privacy Interventions Break Websites," *Proceedings on Privacy Enhancing Technologies*, 2022.
- [86] J. Jueckstock, P. Snyder, S. Sarker, A. Kapravelos, and B. Livshits, "Measuring the Privacy vs. Compatibility Trade-off in Preventing Third-Party Stateful Tracking," in *Proceedings of The Web Conference*, 2022.
- [87] "Secure, Fast and Private Web Browser with Adblocker — Brave Browser," 2024, <https://brave.com/>.
- [88] "sugarcoat/mocks at master · SugarCoatJS/sugarcoat · GitHub," 2021, <https://github.com/SugarCoatJS/sugarcoat/tree/master/mocks>.
- [89] "sugarcoat-paper-dataset/rules.txt at master · SugarCoatJS/sugarcoat-paper-dataset · GitHub," 2021, <https://github.com/SugarCoatJS/sugarcoat-paper-dataset/blob/master/rules.txt>.
- [90] "EasyList," 2024, <https://easylist.to/easylist/easylist.txt>.
- [91] "EasyPrivacy," 2024, <https://easylist.to/easylist/easyprivacy.txt>.
- [92] "GitHub - brave/adblock-lists: Maintains adblock lists that Brave uses," 2024, <https://github.com/brave/adblock-lists>.
- [93] "uAssets/filters at master · uBlockOrigin/uAssets · GitHub," 2024, <https://github.com/uBlockOrigin/uAssets/tree/master/filters>.
- [94] "sugarcoat-paper-dataset/resources at master · SugarCoatJS/sugarcoat-paper-dataset · GitHub," 2021, <https://github.com/SugarCoatJS/sugarcoat-paper-dataset/tree/master/resources>.
- [95] "Using HTTP cookies - HTTP — MDN," 2024, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>.
- [96] "Document: cookie property - Web APIs — MDN," 2024, <https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie>.
- [97] "CookieStore - Web APIs — MDN," 2024, <https://developer.mozilla.org/en-US/docs/Web/API/CookieStore>.
- [98] J. Rautenstrauch, M. Mitkov, T. Helbrecht, L. Hetterich, and B. Stock, "To Auth or Not To Auth? A Comparative Analysis of the Pre- and Post-Login Security Landscape," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.
- [99] J. Rack and C.-A. Staicu, "Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2023.
- [100] "Using Web Workers - Web APIs — MDN," 2024, https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers#importing_scripts_and_libraries.

- [101] “import - JavaScript — MDN,” 2024, <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import>.
- [102] “WebAssembly,” 2024, <https://webassembly.org/>.
- [103] “WebAssembly Concepts - WebAssembly — MDN,” 2024, https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts#porting_from_cc.
- [104] “WebAssembly Web API,” 2025, <https://webassembly.github.io/spec/web-api/index.html#conventions>.
- [105] L. A. Meyerovich and B. Livshits, “ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser,” in *IEEE Symposium on Security and Privacy*, 2010.
- [106] “Introduction — Caja — Google for Developers,” 2021, <https://developers.google.com/caja/>.
- [107] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, “Flow-Fox: A Web Browser with Flexible and Precise Information Flow Control,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.
- [108] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen, “Virtual Browser: A Virtualized Browser to Sandbox Third-Party JavaScripts with Enhanced Security,” in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.
- [109] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, “JSFlow: Tracking Information Flow in JavaScript and its APIs,” in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014.
- [110] P. H. Phung, D. Sands, and A. Chudnov, “Lightweight Self-Protecting JavaScript,” in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, 2009.
- [111] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, “JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications,” in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.
- [112] J. Terrace, S. R. Beard, and N. P. K. Katta, “JavaScript in JavaScript (js.js): Sandboxing Third-Party Scripts,” in *3rd USENIX Conference on Web Application Development*, 2012.
- [113] F. Roesner, T. Kohno, and D. Wetherall, “Detecting and Defending Against Third-Party Tracking on the Web,” in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.
- [114] P. Snyder, L. Ansari, C. Taylor, and C. Kanich, “Browser Feature Usage on the Modern Web,” in *Proceedings of the 2016 Internet Measurement Conference*, 2016.
- [115] J. R. Mayer and J. C. Mitchell, “Third-Party Web Tracking: Policy and Technology,” in *IEEE Symposium on Security and Privacy*, 2012.
- [116] P. Snyder, C. Taylor, and C. Kanich, “Most Websites Don’t Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.
- [117] “Trusted Types - 2.3.4. Default policy,” 2024, <https://www.w3.org/TR/trusted-types/#default-policy-hdr>.
- [118] S. Roth, L. Gröber, P. Baus, K. Krombholz, and B. Stock, “Trust Me If You Can – How Usable Is Trusted Types In Practice?” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.
- [119] “LeakInspector: an add-on that warns and protects against personal data exfiltration,” 2022, <https://github.com/leaky-forms/leak-inspector>.
- [120] “Fathom and Fathom 3.7.3 documentation,” 2019, <https://mozilla.github.io/fathom/>.
- [121] “Error - JavaScript — MDN,” 2023, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error.
- [122] “jQuery,” 2023, <https://jquery.com/>.
- [123] M. Heiderich, C. Späth, and J. Schwenk, “DOMPurify: Client-Side Protection Against XSS and Markup Injection,” in *Proceedings of the 22nd European Symposium on Research in Computer Security*, 2017.
- [124] “GitHub - cure53/DOMPurify: DOMPurify - a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG.” 2024, <https://github.com/cure53/DOMPurify>.
- [125] T. Lauinger, C. Abdelberber, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, “Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web,” in *Proceedings of the Network and Distributed System Security Symposium*, 2017.
- [126] B. Stock, M. Johns, M. Steffens, and M. Backes, “How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security,” in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.
- [127] “JavaScript libraries market share, websites and contacts - Wappalyzer,” 2024, <https://www.wappalyzer.com/technologies/javascript-libraries/>.
- [128] “RequireJS,” 2023, <https://requirejs.org/>.

Appendix A.

A.1. Data Availability

To foster reproducibility, facilitate further research in the area of 3P scripts, and also provide end users with robust and useful privacy enhancing tools, we have open-sourced our system and all plugins we have created (i.e., for SugarCoat, LeakInspector, and ScriptProtect) in a publicly accessible repository [30].

Appendix B.

B.1. Additional Insights

JS inclusion graph. In Figure 4, we show a simplified version of our test page’s JIG. As can be seen, each script node is annotated with additional information. In more detail, each node includes the script’s inclusion URL or method, its source code, whether it has been rewritten or fetched from the cache and if the script executed. Moreover, each script is also annotated with a unique ID (*SUID*) and its parent’s ID (*PSUID*) so as to maintain script inclusion relationships. Most importantly, as shown, StyxJS is able to capture dynamic script injection chains of arbitrary length. In this case, *script1.js* dynamically injects a new script via `eval`, which injects another script via `setTimeout`, which in turn evaluates another script.

JS inclusions. In Table 3 we outline the JS inclusion methods StyxJS captures and compare them with the two original systems we retrofitted on top of it, namely SugarCoat’s underlying PageGraph and ScriptProtect (detailed in § B.4). ScriptProtect has missed several inclusion methods and several others are *partially* captured. For instance, the system captures the injection of new code via `innerHTML`, but *only* if it is utilized to inject markup. However, we found that the same API can be used to set a script’s contents which leads to direct code execution; StyxJS captures such cases as well. We also note that while ScriptProtect is able to attribute the remaining APIs, it only does so because its entire purpose is to prevent *top-level* scripts from introducing further dynamic JS, rendering its `Error().stack` based attribution adequate for its *specific* use case. As extensively outlined in § 5.1, while PageGraph is able to capture all inclusion methods, it cannot *attribute* all of them back to the calling script (e.g., `setTimeout` for string-to-code evaluation), leaving room for trivial bypasses.

TABLE 3: JavaScript inclusion methods covered by different systems. Exposing properties are marked with *.

JS API / property	Condition	PageGraph	ScriptProtect	StyxJS
HTMLScriptElement				
src	-	✓	✓	✓
*text	-	✓	✓	✓
*innerText	-	✓	✓	✓
*textContent	-	✓	✓	✓
append()	Text arg(s)	✓	✗	✓
appendChild()	Text arg	✓	✗	✓
prepend()	Text arg(s)	✓	✗	✓
insertBefore()	Text arg	✓	✗	✓
replaceWith()	Text arg(s)	✓	✗	✓
replaceChild()	Text arg	✓	✗	✓
replaceChildren()	Text arg(s)	✓	✗	✓
Text				
*textContent	Script's child	✓	✗	✓
*outerText		✓	✗	✓
*nodeValue		✓	✗	✓
*data		✓	✗	✓
Element				
*innerHTML	-	✓	☆	✓
*outerHTML	-	✓	☆	✓
insertAdjacentHTML()	-	✓	☆	✓
setAttribute()	Event	✓	☆	✓
Attr				
*value	Event	✓	✗	✓
HTMLAnchorElement				
*href	javascript: URL	*	✗	✓
HTMLAreaElement				
*href	javascript: URL	*	✗	✓
HTMLFormElement				
*action	javascript: URL	*	✗	✓
HTMLButtonElement				
*formAction	javascript: URL	*	✗	✓
HTMLIFrameElement				
*src	javascript: URL	✓	✓	✓
*srcdoc	-	✓	☆	✓
Range				
createContextualFragment()	-	✓	✗	✓
document				
write()	-	✓	☆	✓
writeln()	-	✓	☆	✓
window				
eval()	-	✓	✗	✓
Function()	-	✓	✓	✓
setTimeout()	-	*	✓	✓
setInterval()	-	*	✓	✓
location	javascript: URL	*	✗	✓

✓: captured, ✗: not captured, ☆: partially captured, *: Captured, but not attributed.

Location API. The only JS inclusion method that our system cannot directly override and intercept, are assignments of `javascript: pseudo-URLs` in `window's location` property. This special case of JS execution bears several idiosyncrasies. First, this property cannot be overridden, frozen or deleted, as it is non-configurable. Moreover, while the assignment of a regular HTTP URL would cause the page to redirect elsewhere, `javascript: URLs` do not do so and therefore do not cause any navigation-related events that could otherwise allow us to intercept the JS code *before* it is executed (e.g., `onbeforeunload`).

As such, we leverage the special *default* [117] trusted-type policy (TTP) to robustly handle such cases as well. In more detail, a *default* TTP's `createScript` function will be implicitly called by the browser right *before* assigning the JS URL to the `location` sink. This way, by defining a custom *default* TTP, we can promptly capture, attribute and rewrite the JS injection. However, we also

need to ensure that defining a new *default* policy will not interfere with the page, nor impact the security guarantees of existing TT deployments. To do this, we utilize our background script and, for each main- or sub-frame request, we iterate over its response headers and perform the following actions, depending on the page's usage of TT:

- 1) *Page does not deploy TT*: We add the TT directives with a *default* policy and *create* the policy in the page.
- 2) *Page deploys TT with a default TTP or '*'*: We do not modify the headers, and *wrap* the existing *default* TTP in the page.
- 3) *Page explicitly disables TT OR does not define a default TTP*: We do not modify the headers, nor do we create or modify any TTPs in the page.

For case (1), enabling the TT directives and creating a new *default* TTP does not affect the page, as it is essentially a no-op that only serves our attribution requirements.

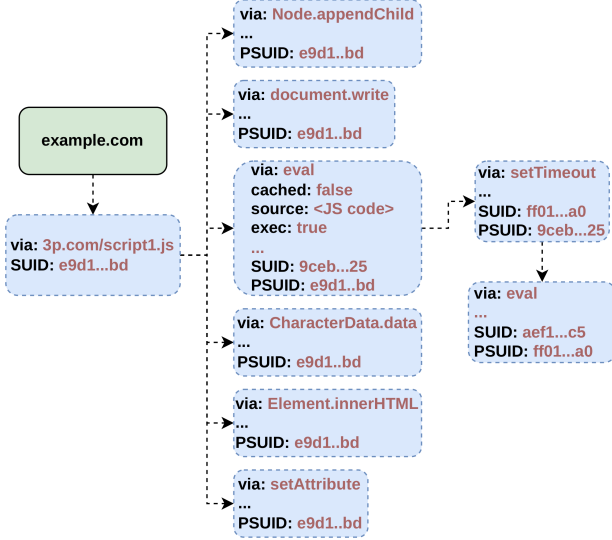


Figure 4: Test page’s JIG. We omit script node details for brevity.

Specifically, our TTP’s `createScript` function determines whether the injection was initiated by a 3P script (using our attribution scheme) and proceeds to rewrite the injected code before returning it to `window.location`, as done in our regular API overrides. For case (2), we override `trustedTypes.createPolicy` so as to intercept the creation of the page’s *default* TTP. Then, we wrap the policy’s `createScript` in our own custom function that attributes the injection, rewrites the code if needed, and eventually calls the original policy. This allows us to maintain compatibility with the page, while also attributing `location` injections. Finally, for case (3), where a *default* policy is not defined, we do not perform *any* actions. This is because dynamically injected 3P code can only be handled by the *default* TTP, since 3P scripts are not aware of and cannot directly use other *named* TTPs [118]. Essentially, this prevents 3P scripts from dynamically injecting new code via any inclusion method. Thus, we refrain from defining a custom *default* TTP, since it is not needed for our attribution, and also because defining one would allow 3P scripts to arbitrarily inject JS code, defeating the original TT deployment’s protection. To the best of our knowledge, no method has been proposed to date for intercepting `location` as a JS inclusion API, and therefore it has not been accounted for by prior work [12], [31].

Error-based attribution. As mentioned in §2, the use of `Error.stack` as the sole means for web script attribution can fall short in identifying dynamically injected 3P scripts. In certain cases however, `Error` can achieve attribution for dynamic scripts as well. For instance, when a top-level script creates a new `script` node and sets its `text` property, its execution will temporarily halt and the new script will be executed. Thus, the `Error` stacktrace will include information about the top-level script too, allowing correct attribution. Nonetheless, if the injected script executes *asynchronously*, after the top-level script has exited, `Error`’s stack is in fact inadequate. This can occur, for example, if the dynamic script encapsulates its functionality in a regular `setTimeout` call (not `string-to-code`). Moreover, this can also occur naturally, e.g., when a top-level script creates a new element

with `innerHTML`, which incorporates an `onclick` or `onload` event listener. In such cases, the event handler will also execute after the top-level script has exited and its information is no longer available in `Error`’s stacktrace. These peculiarities and the complexity they entail in properly tracking dynamic JS injections, as thoroughly explained throughout this work, showcase why robust attribution is not a trivial addition to be made by browsers and also the necessity of a sophisticated system such as StyxJS.

B.2. PageGraph Validation

As mentioned in §4.1, we tried to validate StyxJS against PageGraph in real websites by comparing the scripts each system captures. However, when setting up our experiment, we found that PageGraph can *sometimes* incorrectly capture scripts that *never* execute. For instance, if an `onclick` event handler is set on an element and it does not execute, i.e., the click event is never fired on that element, PageGraph marks it as executed. On the other hand, StyxJS includes it in the JIG but does *not* mark it as executed since its entry code never runs. In a different case, if a text node is appended to a script element which is not yet attached to the DOM and therefore does not execute, PageGraph will *not* capture it, contrary to the previous case. StyxJS behaves consistently, since it also includes the script in the JIG, but does not mark it as executed. Unfortunately, this inconsistent behavior effectively prevents us from performing a correct validation experiment. Moreover, identifying and accounting for *all* cases PageGraph might exhibit such behaviors is out of the scope of this work. Nonetheless, in our test HTML page, where we know precisely what JS code executes and can account for such behaviors, we found 15 distinct dynamic scripts that PageGraph failed to attribute to a 3P, while StyxJS correctly captured all of them. All cases are attributed to PageGraph’s shortcomings, as detailed in §5.1.

B.3. LeakInspector Use Case

Overview. Recently, Senol et al. [29] studied the prevalence of online trackers exfiltrating personal data from HTML forms *before* their submission. Specifically, they measured the exfiltration of e-mail addresses and passwords and found several cases where trackers or session replay scripts extracted both, either intentionally or inadvertently. To aid users in remediating such privacy-invasive practices, they released LeakInspector [119]. This browser extension aims to detect and, optionally, block tracker or 3P scripts from sniffing sensitive elements, pertaining to personally identifiable information (PII), and sending their value to known tracker domains. This is achieved by detecting such input fields in a page, defining a custom `value` getter method on *each* one, and attributing the read operation to the calling script. It also collects the value of all protected fields and inspects outgoing requests to detect and block encoded, hashed and plaintext leakage to trackers.

Limitations & bypasses. LeakInspector relies on naive stack walking for attribution when detecting such read operations, by *solely* leveraging `Error.stack`. As

mentioned in §2, this leads to trivial bypasses through the use of dynamically injected scripts, effectively nullifying its defenses. For instance, a malicious script can still read sensitive input fields’ values by simply injecting a new script through `setTimeout` or `document.write`. Moreover, we have identified several other technical malpractices that enable other types of trivial bypasses. First, as aforementioned, LeakInspector defines a custom `value` method on each PII input field, but does not take precaution to override the corresponding method on the `HTMLInputElement` prototype itself. As such, a malicious script can acquire the prototype’s method and call that instead of LeakInspector’s custom method. Even if this issue was addressed, the system is configured to run on each new document on `document_idle`, after attacker-controlled code has executed. Therefore, a malicious script would still be able to acquire an unprotected reference to `HTMLInputElement`’s `value` method. Similarly, the same can be achieved by acquiring the API reference through a sub-document, as the system has not been configured to run in *all* frames and does not defend against frame polling. In addition, LeakInspector uses a direct reference to `Error.stack`, allowing an evasive script to override the stack trace, remove its information, and avoid attribution. Finally, the system relies on common encodings and hashing algorithms to detect and block PII leakage at the *network level*. As such, a script can utilize any of the above bypass techniques to read PII values and avoid detection, and simply utilize unsupported or nested encodings to send them to tracker domains. As in SugarCoat’s use case (§5.1), we have setup a PoC script that employs all bypass techniques and experimentally verified each one.

StyxJS adaptation. We retrofitted LeakInspector’s approach as a new plugin on top of StyxJS, amounting to 158 LoC. Moreover, we configured the plugin to also inject Mozilla’s Fathom e-mail field detector [120] in each page, as done by the original system. Instead of relying on HTTP request inspection to detect and block PII leakage, which is prone to evasion, we decided to tackle the root cause of the issue. Specifically, we aim to *robustly* prevent 3P scripts from reading sensitive fields’ values in the first place. To achieve this, we initially override the `HTMLInputElement` prototype’s `value` getter method and attribute reads by 3P scripts using StyxJS’ underlying infrastructure. Our override also identifies whether the input field being read constitutes PII, using the same heuristics as [119]. This covers several different types, such as e-mails, passwords and credit card numbers. If the call originates from a 3P script and targets a PII input field, our override will return a predefined dummy value instead of the real one and will also inform the user by highlighting the target element and annotating it with information about the sniffing attempt. In addition, we override the `InputEvent` prototype’s `data` getter and perform the same steps, in order to prevent scripts from registering `oninput` event listeners and gradually constructing the element’s value. We note that this exfiltration method has not been accounted for by LeakInspector. Moreover, our plugin targets *all* 3P scripts, but can be trivially adjusted to only prevent known tracker scripts from reading sensitive values. Finally, we leveraged our PoC script and verified that StyxJS effectively prevents *all* read operations

TABLE 4: Comparison between ScriptProtect and StyxJS.

	ScriptProtect	StyxJS
Domains	112	169
Scripts		
- Blocked	12	22
- Sanitized	102	161
API accesses		
- Blocked	18	140
- Sanitized	719	1,775

that LeakInspector previously missed. We refrain from conducting a real-world comparison between our plugin and the original system, as LeakInspector’s limitations are entirely bypass-related and we do not expect to find scripts actively trying to bypass this specific system.

B.4. ScriptProtect Use Case

Overview. ScriptProtect [12] aims to prevent *benign-but-buggy* 3P scripts from accidentally introducing DOM-XSS vulnerabilities. It achieves this by overriding dangerous HTML sinks that can lead to new JS being evaluated (e.g., `innerHTML`) and sanitizing their input, while APIs that lead to direct JS execution (e.g., `setTimeout`) are entirely blocked. The system offers two modes of operation: for new or relatively short codebases, 1P developers can explicitly use *unsafe* API variants (e.g., `document.unsafeWrite`), essentially calling the original sink. For existing applications, where rewriting calls to unsafe counterparts can be costly, it dynamically disambiguates 3P from 1P calls inside the sink overrides by leveraging the stack trace provided by the `Error` object [121].

Limitations. Unfortunately, for the dynamic 3P attribution mode, ScriptProtect only inspects the top-level stack frame to decide whether a call to a DOM sink originates from a 3P and should be sanitized or blocked. While this makes sense in certain cases, a 3P script could *inadvertently* evade ScriptProtect and introduce a DOM-XSS vulnerability, in case 1P code calls one of its functions. Such 1P calls to 3P code is common practice for many 3P scripts such as analytics, SSO and DOM-manipulation libraries [1], [3], [4], [122]. Moreover, ScriptProtect has overlooked *several* JS inclusion methods (see Table 3) and in some cases only partially captures them – we have found cases where their interception fails and new JS code is executed. This further allows 3P scripts to unintentionally circumvent its defenses and introduce DOM-XSS bugs. We have experimentally verified these limitations.

StyxJS adaptation. To retrofit ScriptProtect on top of StyxJS, we created a plugin (110 LoC) implementing all their dangerous sinks’ overrides, as well as the ones that they missed, and adjusted them to use our 3P attribution interface. It also injects `DOMPurify` [123], [124] in each page for the markup APIs’ sanitization step, as per the original system.

To showcase StyxJS’ ability to robustly and conveniently retrofit existing pipelines, but also combine them or introduce new ones, we decided to target outdated JS libraries for our experiment, as they are more likely to be vulnerable, an issue highlighted by prior studies [12], [125], [126]. To that end, we slightly modified the original

system to target *specific* 3P scripts, and modified DOM-Purify to use references to *original* APIs, as we found cases where ScriptProtect would wrongfully attribute such calls to 3P scripts. To construct a dataset of outdated JS libraries we referred to [125], which reported the 30 most popular libraries of the time. We also gathered the 20 most currently-popular libraries (according to Wappalyzer [127]). Next we removed duplicates, NodeJS modules (which are irrelevant in our context), and well-known, functionality-critical libraries that might entirely break a website if tampered with (e.g., jQuery [122], RequireJS [128]); this left us with 32 JS libraries. We stress that excluding these libraries is only necessary for this specific use case and does not affect StyxJS’ overall applicability.

Next, we visited each library’s repository and gathered the version numbers that have not been updated in over two years. We also identified URLs commonly used to include each library in the wild, based on repository documentation and relevant StackOverflow questions. Based on these URLs and versions, we compiled a list of regex-based filter rules that we fed into our plugin and the original system, in order to capture these scripts specifically. We note that our dataset is rather limited, as we might have missed versions, URLs and even entire libraries. However, our goal is not to study the use of such libraries, but to compare StyxJS’ effectiveness over ScriptProtect in a realistic use case; as such, we believe the compiled dataset serves our goal. Finally, we visited the landing page of the top 3K domains with both systems and recorded protected API accesses; this process yielded 169 domains that included one of the outdated libraries.

As shown in Table 4, StyxJS offers a significant improvement over ScriptProtect as it blocks approximately 8 times more and sanitizes 2.5 times more API calls, while covering 57 more domains where at least one protected API was called. StyxJS also captures 10 and 59 more scripts which called at least one blocked or sanitized API, respectively. Regarding page load times, we observe that our plugin takes up to 0.07-0.12 and 0.7-0.73 seconds more for 50% and 90% of the domains respectively, for two subsequent page loads, once again highlighting the practicality of our approach.

To investigate possible breakage due to StyxJS, we analyzed 50 random domains with both systems. We found 19 cases where the original system threw an exception and caused minor breakage, such as appearance issues, submitting cookie policies, playing videos, and breaking the search functionality. Moreover, in five of these domains, ScriptProtect caused their navigation menus to not load properly, effectively disabling access to other pages. In contrast, we did not encounter any such issues with StyxJS in *any* of the domains.