Unleash the *Simulacrum*: Shifting Browser Realities for Robust Extension-Fingerprinting Prevention

Soroush Karami[*†], Faezeh Kalantari[*±], Mehrnoosh Zaeifi[±],
Xavier J. Maso[±], Erik Trickel[±], Panagiotis Ilia[†], Yan Shoshitaishvili[±], Adam Doupé[±], and Jason Polakis[†]

[†]University of Illinois at Chicago, {skaram5, pilia, polakis}@uic.edu
[±]Arizona State University, {faezeh.kalantari, mzaeifi, xmaso, erik.trickel, yans, doupe}@asu.edu

[*]Joint first authors.

Abstract

Online tracking has garnered significant attention due to the privacy risk it poses to users. Among the various approaches, techniques that identify which extensions are installed in a browser can be used for fingerprinting browsers and tracking users, but also for inferring personal and sensitive user data. While preventing certain fingerprinting techniques is relatively simple, mitigating behavior-based extension-fingerprinting poses a significant challenge as it relies on hiding actions that stem from an extension's functionality. To that end, we introduce the concept of *DOM Reality Shifting*, whereby we split the reality users experience while browsing from the reality that webpages can observe. To demonstrate our approach we develop Simulacrum, a prototype extension that implements our defense through a targeted instrumentation of core Web API interfaces. Despite being conceptually straightforward, our implementation highlights the technical challenges posed by the complex and often idiosyncratic nature and behavior of web applications, modern browsers, and the JavaScript language. We experimentally evaluate our system against a state-of-the-art DOM-based extension fingerprinting system and find that Simulacrum readily protects 95.37% of susceptible extensions. We then identify trivial modifications to extensions that enable our defense for the majority of the remaining extensions. To facilitate additional research and protect users from privacy-invasive behaviors we will open-source our system.

1 Introduction

The modern web has permeated numerous aspects of our everyday lives and, thus, reshaped how we conduct many sensitive and critical operations. At the heart of users' online experience lie web browsers, mediating a wide range of sensitive communications and activities. Unfortunately, while browsers are a portal to limitless potential, their rich set of features and complex functionality can also enable or facilitate privacy-invasive behaviors [12, 26, 29, 34, 58]. As a result, in recent years web tracking has garnered significant attention from researchers and practitioners alike.

Due to the stateless nature of the HTTP protocol, web tracking has traditionally relied on the presence of cookies. However, with users becoming more privacy-cautious and browsers continuing to deploy anti-tracking defenses that hinder cookie-based tracking [20, 60, 65], trackers have also evolved accordingly. In fact, a wide range of techniques have been demonstrated by researchers or found in the wild; from "supercookies" and "evercookies" (e.g., using HSTS policies [59], internal storage [7], or favicons [53]) to DNS-based trickery [9, 17, 28], these techniques highlight the feasibility and creativity of tracking techniques that bypass existing defenses. While browsers may gradually adapt and prevent such emerging techniques, one of the most alarming modern approaches is that of browser fingerprinting. Prior research has demonstrated many browser fingerprinting vectors targeting underlying system and hardware characteristics [8, 11, 15, 19, 21, 22, 30, 31, 39–42, 63]. Additionally, a more recent line of research has focused on how installed browser extensions can be used for fingerprinting and tracking, as each user will install a unique set of browser extensions. To make matters worse, installed extensions carry semantic information that can be used to infer sensitive user traits such as religion, sexual orientation, and medical issues [27].

Prior studies have proposed mitigations that target different aspects of extension fingerprinting, namely preventing techniques that target Web Accessible Resources (WARs) [48], stylesheets (CSS) [32], and node attributes [61]. However, no existing defense can effectively prevent DOM-based fingerprinting [27], since many extensions intentionally modify pages in varied and diverse ways and these changes can be uniquely identifiable. Essentially, the root cause of behavior-based extension fingerprinting is that any JavaScript running in the context of the web page can see all the changes and modifications that the installed extensions make to the page.

In this paper, we address the root cause of this robust fingerprinting technique, by approaching the problem from a fundamentally different perspective. We propose the notion of *DOM Reality Shifting*, wherein we split the reality that a user experiences when browsing a page from the reality that the page can actually observe. By separating the page's DOM, the user

sees the changes that an extension makes to the page, while the page's JavaScript cannot see those changes. In more detail, we create a *Parallel DOM* in addition to the *User DOM*, and mediate access such that extensions query, edit and interact with the User DOM while page JavaScript is limited to interacting with the Parallel DOM. While DOM Reality Shifting is conceptually straightforward it is fundamentally effective against DOM-based behavior fingerprinting, yet correctly implementing it requires handling a myriad of JavaScript idiosyncrasies, corner-cases and real-world complexities that can undermine the security of such a system or the functionality of web applications. To demonstrate the feasibility of our approach we develop a prototype extension called Simulacrum that implements DOM Reality Shifting *without the need to change the browser*, through a targeted instrumentation of Web API interfaces.

To assess the practicality of our approach we experimentally evaluate Simulacrum across multiple dimensions. First, we demonstrate our system's effectiveness by deploying it against a state-of-the-art automated DOM-based fingerprinting system [27]. Out of 5,793 fingerprintable extensions our system effectively hides the presence of 95.37% of the extensions. We then measure the overhead introduced by our defense and find that it is less than 390ms for half of the websites and $\sim 895ms$ on average. Finally, we manually assess how Simulacrum affects extensions' and websites' functionality and find that all extensions remain unaffected, while major and minor breakage occurs in 12% and 10% of sites respectively. To avoid breakage, our extension's users can allowlist trusted sites.

In summary, our research contributions are:

- We introduce the concept of *DOM Reality Shifting*, which fundamentally addresses the root cause of DOM-based browser extension fingerprinting. Simulacrum, our prototype extension, significantly limits extension fingerprinting without modifying the browser, thus allowing for immediate and widespread adoption.
- We experimentally evaluate Simulacrum's defensive and performance impact, and demonstrate that our system effectively protects extensions while incurring a negligible performance overhead and limited website breakage.
- We present guidelines for extension developers that allow them to eliminate problematic fingerprintable behaviors without affecting the extension's functionality. These guidelines require straightforward changes that are trivial to implement, and can contribute to completely eliminating DOM-based fingerprinting.
- To further reproducibility in science, we will open-source our system as well as the list of all domains and extensions (including versions) used in our experiments.

2 Background and Threat Model

Browser fingerprinting relies on extracting unique attributes of the user's browser and device. Among those, the list of installed browser extensions can be coupled with other information to build reliable fingerprints. Browser extension fingerprinting is indeed a real-world threat, as LinkedIn was found trying to detect 38 different extensions [45].

Extensions rely heavily on the JavaScript language and customize web pages by modifying the page's DOM. An invasive web page can observe this behavior (i.e., the modifications made to the page's DOM) and use it to construct a set of behavior-based fingerprints (i.e., signatures). When a user visits the page, the invasive page can use the fingerprints and the changes made to determine the extensions the user has installed. Since the extension modifies the page's DOM, effectively hiding the extension's behavior from the page without breaking the page's functionality is a challenging problem that prior defenses failed to truly address [27].

Threat model. We assume the attacker controls a specially crafted web page or iframe that implements DOM-based fingerprinting to uncover the extensions installed in the user's browser. In a nutshell, DOM-based fingerprinting relies on JavaScript that leverages the Web API [36] for direct read and write access to the DOM; by observing and interacting with it, the attacker can deduce which extensions are installed based on how the extensions modify the DOM. We note that social engineering attacks that trick the user into divulging which extensions they have installed (e.g., following a similar strategy to [64]) and side-channel attacks (e.g., timing-based) are considered out of scope for our defense.

Browser extensions. Users install extensions in their browsers to expand the browser's functionality and improve their browsing experience [16]. Extensions offer such features via a bundle consisting of HTML, CSS, JavaScript code, and a configuration file called the `manifest`.

Security model. To reduce the threat of malicious pages compromising browser extensions [43, 54], browsers employ privilege separation for extensions. Specifically, extensions' `background scripts` have powerful privileges but do not have access to the DOM, and extensions use `content scripts` to access the DOM, which lack the extension's full capabilities. Although `content scripts` have full access to the DOM, they do not run in the `website scripts'` execution environment. That is, neither `website scripts` nor `content scripts` are able to directly access one another because each runs in its own isolated world, while sharing the same DOM.

DOM-based fingerprinting. Due to the sharing of the DOM across the isolated worlds, invasive pages can construct DOM-based fingerprints by observing the extension's modifications to the page's DOM [57]. Prior work [27] has demonstrated how an attacker can automatically construct an extension's fingerprint by capturing the extension's DOM modifications that alter specific fields (e.g., username, password), DOM elements (e.g., images), or text keywords. Then, when a victim visits a specially crafted web page that contains a comprehensive set of elements and features, the fingerprinting framework can compare the DOM modifications to the previously captured fingerprints to identify the extension.

JavaScript uses prototypes to share properties (values and functions) and define hierarchical relationships between objects. Prototype-based languages do not use classes to generalize the characteristics of a set of objects. Instead, any object can share its properties with others as part of its prototype chain [35].

Prototypal inheritance. Effectively, the prototype chain describes the inheritance hierarchy of an object, and is used when code tries to access an object's attribute. During execution, JavaScript looks for the attribute within the object itself, and returns the attribute if found. Otherwise it recursively searches the prototype chain until it finds (and returns) the attribute or reaches the end of the chain (and returns `undefined`).

Function overriding. JavaScript can dynamically update properties of existing objects. Programmers leverage this behavior to *wrap* existing functionalities and customize their behavior. Additionally, JavaScript can override the `getter` and `setter` functions of a prototype's properties to customize prototype modifications themselves. In the Appendix we provide code detailing how Simulacrum overrides prototype values and functions, including getters and setters.

3 DOM Reality Shifting with Simulacrum

We present Simulacrum, a novel countermeasure against DOM-based extension fingerprinting. Here, we provide an overview of our design and the techniques employed by our system for hiding the presence of extensions from malicious or invasive webpages. The core strategy behind our defense is to create a *split reality* between what a user experiences when browsing a page and what the page can actually observe. Specifically, Simulacrum hides the artifacts created by extensions modifying the DOM by creating an alternate reality for the page that does not include any of the DOM modifications of installed extensions. The user's reality includes not only the page but also all DOM modifications performed by extensions, thus offering users the same browsing experience with enhanced privacy protections.

DOM reality shifting. Simulacrum places the webpage JavaScript in a parallel reality that does not contain extensions' DOM fingerprints by creating a simulacrum of the User DOM that omits the extensions' DOM-fingerprints, the *Parallel DOM*, and routing all the webpage's DOM requests to the Parallel DOM. To isolate the altered reality, Simulacrum prevents the webpage from accessing the *User DOM*, the browser-created DOM that the user sees and that contains all the modifications made by the webpage and the user's extensions. In addition, Simulacrum maintains the webpage's altered reality by mediating all access to and ensuring consistency between the two DOMs.

Modification mirroring. Simulacrum replicates changes between the User and Parallel DOMs depending on the origin of the request. For modifications originating from the website or user input, Simulacrum replicates the modifications in the Parallel DOM. However, Simulacrum limits modifications made by an extension to the User DOM.

Query routing. Simulacrum automatically routes queries to either the User or Parallel DOMs depending on the query's origin. When an extension requests access to the DOM, it automatically accesses the User DOM. However, when the webpage accesses the DOM, our system seamlessly routes the request to the Parallel DOM. As a result, the website is unable to detect DOM-fingerprints left behind by extensions because in the website's DOM reality the extension's changes simply do not exist and, thus, no fingerprintable artifacts are left behind by the extensions' functionality.

Interception. Simulacrum controls all requests and performs the necessary replication functionality by wrapping the DOM interface with an intelligent router. We achieve this by being the first script to execute and overriding the appropriate DOM interfaces before the website's JavaScript executes.

4 System Implementation

This section covers Simulacrum's implementation details and the technical challenges that it addresses to effectively achieve DOM Reality Shifting. Here, we use the following notation: `e` refers to a JavaScript object of type (i.e., respecting the interface) `Element`, and `n` refers to a JavaScript object of type `Node`.

4.1 Primitives

Creating the Parallel DOM. To preserve compatibility, the Parallel DOM created and maintained by Simulacrum must include every aspect of the User DOM *except* for the DOM modifications introduced by the user's extensions. Simulacrum instantiates the Parallel DOM by cloning the User DOM after it receives the `DOMContentLoaded`, which is triggered by the browser after it finishes loading and parsing the page's HTML and without waiting for the stylesheets, images, and subframes to load. Simulacrum maintains the Parallel DOM throughout the page's life cycle by continually (1) ensuring the Parallel DOM's consistency with the User DOM and (2) preventing the propagation of extension-driven changes to the Parallel DOM.

Equivalent elements. Propagating DOM modifications between the User and Parallel DOMs requires a method for accurately mapping equivalent elements across them. We create this mapping using `id` attributes, which are unique within a `Document` [13]. That is, for an element with an `id` value of `"node_id"` in the User DOM, Simulacrum calls `parallelDOM.getElementById("node_id")` to access the equivalent element in the Parallel DOM. When DOM elements lack an `id` attribute, we assign them a unique random value before creating the Parallel DOM. In addition, we handle elements created after the Parallel DOM is instantiated. To handle this, Simulacrum wraps functions that create new DOM elements (e.g., `document.createElement()`) with logic that assigns them a unique `id`.

Simulacrum implements a function, `getEquivalent(e)`, to return the element equivalent to `e` from the other DOM. First, the function checks whether the provided `e` originates from the User DOM or Parallel DOM; if `e` does not originate from either (i.e., `e` is not attached to either DOM), the function returns `e`. Next, if the `id` of `e` exists in both DOMs, then the function returns the element with the same `id` (i.e., the equivalent element) from the non-originating DOM. Otherwise, the function returns `null`. In practice, websites might use the same ID for multiple elements. If the `getEquivalent()` function detects non-unique IDs, it uses `querySelectorAll("[id='node_id']")` to get the list of all elements with `id='node_id'`. Then, if the element is the i-th element with `node_id` in one DOM, the equivalent will be the i-th element with `node_id` in the other DOM. For simplicity, in the remainder of this paper we will only mention `getElementById` for finding equivalent elements.

Cloning. The standard JavaScript method for making node copies, `cloneNode()`, does not copy custom properties or the source object's event listeners. To fix the limitations of `cloneNode()`, Simulacrum uses a custom `deepClone()` function. After making a clone with `cloneNode()`, `deepClone()` uses `Object.keys()` to get all the custom properties and transfer them one by one to the cloned node.

Unfortunately, JavaScript does not provide an API for accessing the list of all event listeners for each node. The `Element` and `HTMLElement` interfaces provide APIs for setting and getting event listeners (such as `HTMLElement.onclick`), and `deepClone()` uses these APIs to check for and clone event listeners on the node. However, JavaScript does *not* provide a direct way for accessing the event listeners that are set using `EventTarget.addEventListener()`. Simulacrum overrides this function to intercept all invocations of it and collect (and, later, clone) the event listeners that are set for each node.

Referencing original functions. Simulacrum needs access to the original versions of the functions it overrides to affect its operations on the DOMs. Before overriding them, we store the methods and functions of these interfaces in a hash table (accessible only by the Simulacrum content script).

4.2 DOM-accessing APIs

The DOM API is incredibly complex, comprised of hundreds of interfaces and thousands of functions. However, we only need to override APIs that have either read or write access to the DOM; Simulacrum wraps the original functions with logic to restrict the webpage's access to the User DOM.

Read access. Simulacrum wraps functions to ensure that JavaScript code run by a webpage reads from the Parallel DOM. For example, `Element.querySelector()` has read access to the DOM, and we override it to query the parallel DOM.

Write access. Simulacrum wraps functions with write access. The write access wrappers apply the necessary DOM modifications that originate from website scripts to both DOMs and ensure that they are synchronized (i.e., consistent). For example, `Element.append(<newElement>)` has write access and the wrapper appends `<newElement>` to both DOMs.

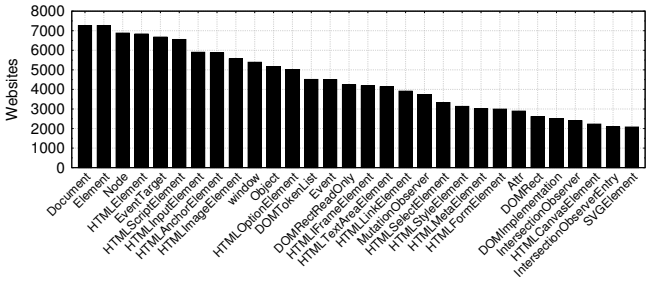

Figure 1: Most prevalent prototypes across top 10k websites.

Identifying APIs. While `Document`, `Element` and `Node` are obvious items to be added to our overriding list, an automated and systematic approach was necessary for identifying other commonly used interfaces to read from or write to the DOM. We developed a Chrome extension, called `VisibleJS`, that records all function invocations that occur during the execution of JavaScript code on a page. `VisibleJS` overrides all the functions implemented by JavaScript and logs the functions each time the page's code invokes them, using the code shown in Listing 3 (Appendix). For each function invocation, `VisibleJS` analyzes the arguments, returned value, and receiver. If at least one of those is an instance of `Node` or a DOM-related object (e.g., `DOMTokenList` and `DOMStringList`), then it records the function as one with access to the DOM.

Inspired by prior approaches on quantifying the prevalence of browser features [52], we chose to use the 10K most popular sites (according to the Alexa ranking) to determine commonly used interfaces. We use Selenium, ChromeDriver, and `VisibleJS` to capture the functions used by each site that access the DOM. Figure 1 shows the number of websites that use at least one function of the 30 most popular interfaces. Overall, our experiment uncovered a total of 135 interfaces (and 1,532 functions) that interacted with the DOM among the top 10K websites. However, as one might expect, not all interfaces are equally popular, as more than half (70) of the interfaces are used by less than 1% of the websites. Balancing the popularity of the functions and the effort required for manually overriding each function, we decided to override the 75 most popular interfaces. In addition, we override all interfaces that inherit from the `Node` interface (some of them are not used by the top 10K websites). In total, Simulacrum overrides 156 interfaces.

VisibleJS lead us to certain interesting findings, which shed light on the intricacies and complexities of the DOM API's interfaces. For instance, we found it surprising that `XMLHttpRequest` interacts with the DOM. `XMLHttpRequest` is a commonly used interface for interacting with servers. However, `XMLHttpRequest`'s `responseXML` argument returns an object that implements the `Document` interface and can then be appended to the DOM.

4.3 Function Overriding

Here, we detail the processes Simulacrum employs for overriding read and write functions available through the DOM API.

Categorization. We first categorize functions depending on their type of DOM interactions. Table 5 (Appendix) displays different function examples for each category with their overridden counterpart. We provide simplified versions of the actual call invocations throughout the text to improve readability.

Simple getter. This category of functions returns static information about the DOM. For example, `document.getElementsByTagName('div')` returns all the `div` elements of the DOM, and `e.hasAttribute('src')` returns `true` if the object `e` has a `src` attribute. Simulacrum overrides these functions so they return the result of their execution on the parallel DOM. Accordingly, it replaces the examples given above with `parallelDOM.getElementsByTagName('div')` and `parallelDOM.getElementById(e.id).hasAttribute('src')`.

Active getter. This category of functions cannot be executed on the Parallel DOM. For example, `e.scrollTop` returns the number of pixels that the element's content is scrolled vertically. However, since the Parallel DOM is not visible to users, this value will always be zero for any element tested. As such, for this type of function we first check if `e` has an equivalent in the Parallel DOM (thus ensuring that it was not the product of an extension), and then run this function on the User DOM to obtain the appropriate value that should be returned.

As another example, `document.activeElement` returns the element in the DOM that currently has focus. Since the Parallel DOM is not visible, its elements cannot have focus. Thus, we run this function on the User DOM and return the equivalent element on the Parallel DOM. If the element that is currently focused was created by an extension, we return its first ancestor not added by an extension (i.e., its first ancestor that has an equivalent in the Parallel DOM).

Simple setter. Functions in this category modify the structure of the DOM or a node's attribute. To replicate such DOM modifications, we run the function on both the function receiver and the equivalent of the receiver. For instance, when a webpage's script invokes `e.innerHTML = "text"`, the wrapper checks that the `id` exists in the Parallel DOM. If it does exist, then the wrapper sets the `innerHTML` of `e` and its equivalent object. For example, if the Parallel DOM owns `e`, then the wrapper runs `e.innerHTML = "text"` and `document.getElementById(e.id).innerHTML="text"`.

Active setter. Functions in this category do not directly modify the DOM. As such, we simply direct these to the User DOM. For example, `e.requestFullscreen()` results in the element being displayed in full-screen mode. The wrapper for this function first checks the element's owner document. If the owner document is the Parallel DOM, we find its equivalent element in the User DOM and run this function on it. Otherwise, we simply execute the function.

Forwarding arguments. It is important to note that any of the functions in the mentioned categories might have arguments. Almost all arguments can simply be passed to the corresponding functions. However, if an argument is a DOM object (an object of type `Node`, `Element`, etc.), the wrapper might need to invoke the function with the equivalent of arguments.

For instance, `parentNode.insertBefore(newNode, refNode)` inserts the `newNode` before the `refNode`, which is a child of `parentNode`. In this case, Simulacrum uses the overriding strategy used for the *simple setter* category. That is, it runs this function two times, once for `parentNode` and once for its equivalent one. Unlike the *simple setter* category, the `insertBefore` function receives two DOM objects as arguments and, as a result, the wrapper locates and passes the arguments `equivalentNewNode` and the `equivalentRefNode`. While the wrapper uses `getEquivalent(refNode)` to find the equivalent version of `refNode`, the `newNode` does not have an equivalent node because it is not connected to a DOM yet. Thus, the wrapper creates a `deepClone()` of `newNode` and uses that for `equivalentNewNode`.

Interfaces of the DOM API. The DOM API is structured around different *Interfaces* effectively grouping JavaScript objects with common state and behavior together. Note that all objects in JavaScript are instances of the `Object` class, which is on the top of the prototype chain. For the interested reader, Figure 6 (Appendix) shows a partial representation of the DOM API with some of its interfaces.

Node interface. Every node of the DOM tree is represented by an object of type `Node`, which also includes any interfaces inheriting from it (notably `Attr`, `Document`, `Element` [13]). The `Node` interface describes properties (e.g., `nodeName`, `parentNode`) and methods (e.g., `cloneNode()`, `normalize()`) that are shared amongst all DOM objects [13]. We wrapped 14 properties and 15 methods in `Node`'s prototype.

Document interface. This interface represents the result of parsing the page, and grants access to the DOM. It describes the common properties (e.g., `title`, `bgColor`) and methods (e.g., `querySelector()`, `createElement()`) for any kind of document [13]. We note that this interface exposes properties for setting event handlers (e.g., `onclick`,) [13]; these are in the active setter category, and we set these event listeners in the User DOM. In practice, the User DOM fires the events because the Parallel DOM is in the background. During the execution of callback functions from event handlers, as with any other JavaScript function, Simulacrum applies modifications to both the DOMs. The `Document` interface also allows the creation of specific DOM objects [13]. In particular, Simulacrum needs to wrap the `createElement()`, `createElementNS()`, and `createDocumentFragment()` methods to forcefully assign a value to the `id` attribute of new elements created through this interface (see §4.1). Finally, `createTreeWalker()` and `createNodeIterator()` instantiate objects that help JavaScript perform DOM traversal [13]; Simulacrum uses the *simple getter* strategy to prevent them from reading

the User DOM. More specifically, Simulacrum's wrapper for `document.createTreeWalker(root)` converts it to `parallelDOM.createTreeWalker(equivalentRoot)`. This limits the webpage's access to the Parallel DOM. In the end, we override 264 properties and 40 methods.

Element interface. All elements in the DOM inherit a set of methods and properties common to all types of elements from `Element`. As shown in Figure 6, each element object derives from either `HTMLElement` or `SVGElement`. Both of them count several descendant interfaces: 72 for `HTMLElement`, and 71 for `SVGElement` [13]. `HTMLElement` serves as the base interface for HTML elements. Some elements directly derive from this interface, while others implement this interface using another interface that inherits it. For example, `<footer>` elements directly implement `HTMLElement`, while `<video>` elements implement `HTMLVideoElement`, which inherits from `HTMLMediaElement`, which inherits from `HTMLElement`. Additionally, all the SVG elements of the SVG language inherit the `SVGElement` interface. Simulacrum wraps all the methods and properties of the `Element` interface and its descendants with an appropriate overriding strategy. To that end, we manually analyzed all 1,532 methods and properties, so as to choose the correct overriding strategy for each case.

Observer interfaces. The Web APIs provide different observers allowing JavaScript executed in the browser to observe, be notified, and react (using callback functions) to updates in the state of DOM objects. Simulacrum needs to override three observers: `ResizeObserver`, `IntersectionObserver` and `MutationObserver` [13, 14, 66]. Listing 4 (Appendix) gives examples that demonstrate the three observers.

ResizeObserver interface. Objects of this interface monitor changes to the size of elements (size may change for various reasons, such as changes in the size of the browser window). The `ResizeObserver`'s constructor receives a `callback` function. Each time the size of an observed element changes, the browser notifies the observer by executing the provided *callback* function. The `observe` method starts the observation of the specified element and identifies the target of interest. We use the active setter strategy for overriding the observe function. That is, Simulacrum prevents a webpage from observing an element that does not exist in the Parallel DOM.

IntersectionObserver interface. This interface provides the ability to observe changes in the intersection of a target element with another object called the *root*. The `IntersectionObserver`'s constructor receives a callback and an optional configuration variable which, when provided, defines the `root`. The root descends from a specific target or the viewport. If the constructor does not include the configuration argument, the observer assigns the viewport as the root. Simulacrum wraps two functions: first, the `IntersectionObserver`'s constructor to configure a proper root for its object. To do so, the wrapper checks that the configuration argument contains a root and that it exists in the Parallel DOM. If so, it uses `userDOM.getElementById(root.id)` to replace the root with the one from the User DOM. Second, we wrap the `observe` function using the active setter strategy, which prevents pages from using this API to detect extensions.

MutationObserver interface. This interface allows scripts to observe changes made to the DOM. The `MutationObserver`'s constructor receives the callback function. The `observe` method receives the target node and the observer configuration. The configuration determines the types of DOM changes the observer will react to. For example, by passing a configuration with `childList: true` and `subtree: true`, the observer triggers the callback for DOM modifications that change the list of children attached to the target node or the nodes in its subtree. The list of children changes through the addition or removal of nodes. The `observe` wrapper uses a slightly more complicated version of the active setter strategy. The `MutationObserver` invokes the callback when it observes modifications to the target node and the target node's descendants. For instance, if the target is the `<html>` element, the observer receives notifications for new elements that the browser appends to any location in the page. Subsequently, the wrapper adds functionality to avoid invoking the callback when an extension mutates the DOM. To that end, Simulacrum overrides `MutationObserver`'s constructor to modify the incoming callback. The callback modifications cause it to skip DOM mutations originating from an extension and only invoke the original `callback` for those DOM changes that originate from the webpage's scripts. Listing 5 in the appendix shows the wrapper code. The wrapper replaces the original callback with `newCallback`. The `argument[0]` for `newCallback` is an array of mutation records observed by the `MutationObserver`. The wrapper uses the `filterMutations()` function to filter out the mutations caused by an extension. If any mutations remain, the wrapper invokes the original `callback` with the remaining webpage-only mutations.

The `filterMutations()` function operates based on the attributes of each mutation's record. For example, if an extension appends a new `<p>` element to the User DOM, `mutation.type` will be `"childList"`, the value of `mutation.target` will be the reference to the parent of the `<p>` element (i.e., the element that it has been appended to) and `mutation.addedNodes` refers to the `<p>` element. The `filterMutations()` function then checks the Parallel DOM for its equivalent of `<p>` and `mutation.target`. In this example, since `<p>` is injected by an extension and is not a child of the Parallel DOM's version of `mutation.target`, the function filters out the current mutation from the mutation array.

In another example, if one of the webpage's script modifies the `src` attribute of an `<img>` element, the `mutation.type` will be `"attributes"`, the `mutation.target` will be a reference to the `<img>` element, and the `attributeName` will be `"src"`. Based on `mutation.type`, the `filterMutations()` investigates the attributes by comparing the `src` attribute of the `<img>` element with the `src` of its equivalent from the non-originating DOM. In this case, it does not filter out the

current mutation out of the mutation array because the `src` value from both DOMs are equal. Thus, the wrapper invokes the original `callback` with the remaining mutations.

Other interfaces. We use an element's `id` to find equivalents in the two DOMs. Objects that do not implement the `Element` interface and lack an `id` require a different approach; the key observation for these objects is that they are connected to an element. Thus, to find their equivalent we leverage the element they are connected to. E.g., `element.classList` returns an object that implements the `DOMTokenList` interface, which has functions like `add()` for modifying the object. When the page calls `tokenList.add("name")`, we also call `equivalentTokenList.add("name")` to propagate this to the other DOM. Since `tokenList` does not implement `Element`, we use `element` to find the equivalent, and `equivalentElement.classList` to set `equivalentTokenList`. Accessing the connected element differs for different non-element objects. Some non-element objects implement properties, for example `ownerElement`, `parentElement`, or `parentNode`, which provide direct access to the connected element. However, other objects do not provide a direct mechanism for accessing their connected element (e.g., `DOMTokenList`). For indirect cases, Simulacrum automatically adds an owner element property to these objects the first time they are called. For example, `element.classList.add("name")` modifies the object that is returned by `element.classList`. Indeed, the call results in two API calls: 1) `element.classList` returns a `DOMTokenList`, 2) `tokenList.add("name")` adds the string to the list. In the wrapper function of `classList`, the wrapper sets `element` as the owner element of the returned `DOMTokenList` object. Therefore, in the `add()` function, Simulacrum can get the owner element, which it can then use to find the owner of non-element's equivalent.

It is important to emphasize that knowing the owner element is not sufficient for finding the equivalent object. In addition to the owner element, Simulacrum needs to know the function that was called for getting this object. For example, the `Element.classList`, `HTMLLinkElement.relList`, and `HTMLIframeElement.sandbox` properties return an object that implements the `DOMTokenList`. After finding the equivalent owner element, we need to run the proper function to get the equivalent non-element object. To do this, the wrapper function stores the API call for accessing the object, which the wrapper later uses to get the equivalent non-element object.

4.4 Additional Security Precautions

During our design and development process we accounted for potential attacks against Simulacrum and incorporated appropriate defenses to prevent circumvention of our protections.

IIFE. As mentioned in §4.1, Simulacrum stores the original version of all the interfaces in a hash table. If attackers are able to access the original interfaces, they can replace the overridden functions with the original ones and access the User DOM. To prevent this attack, Simulacrum leverages Immediately-invoked Function Expressions (IIFE) for isolating functions and variables. That is, we encapsulate our system in an anonymous function that the JavaScript environment executes immediately. In essence, this allows us to define our logic within another function while the inner functions have access to the variables in the outer function's scope. Using this approach Simulacrum has access to the global scope of the webpage but prevents the webpage's JavaScript code to access objects like the hash table of original interfaces that are used by Simulacrum.

Position-based attacks. Simulacrum executes *active getter* functions on the User DOM. While they cannot be executed on elements injected by extensions, they can be used to get the position of other elements. Since injected elements can affect other elements' position, attackers could potentially identify installed extensions by observing changes in the positions of elements. To prevent this, Simulacrum adds `<div>` and `<span>` elements with random sizes at different locations throughout the User DOM. Since these do not exist in the Parallel DOM they are not visible to the websites' scripts and, thus, attackers cannot infer whether changes to the positions of elements occur due to injections by extensions or the random noise introduced by our system. It is important to note that these noise elements do not really impact the user's browsing experience as they simply introduce empty spaces around other elements (see Figure 8 in the Appendix).

While adding noise prevents attacks against extensions that inject elements, this might not work for extensions that remove elements. If an extension removes an element from the User DOM, the element will not be removed from the Parallel DOM; however, from the changes in the location of other elements an attacker might be able to infer that extension's presence. There are two categories of extensions that remove elements from websites. The first category is extensions that block scripts which are responsible for creating elements (e.g., ad-blockers). The second category includes extensions that explicitly remove elements which are already connected to the DOM. While Simulacrum cannot observe the behavior of the first category, the behavior of the second category is DOM-based and can be detected by Simulacrum. To that end, when an element is removed by an extension, Simulacrum replaces the removed element with another element of the same size; this prevents changes to the location of other elements. We have experimentally found that such extensions are very rare (only 59 fingerprintable extensions from §5 remove an element). As such, Simulacrum only activates this defensive mechanism against element removal when the user has one of these 59 extensions installed; in our prototype, this is done through a hard-coded list but can easily be enforced based on a list that is fetched from a server (similarly to EasyList [2]).

Order of execution. If the attacker's code executes before Simulacrum, they can access the original interfaces. To prevent this, Simulacrum injects its script into the webpage's

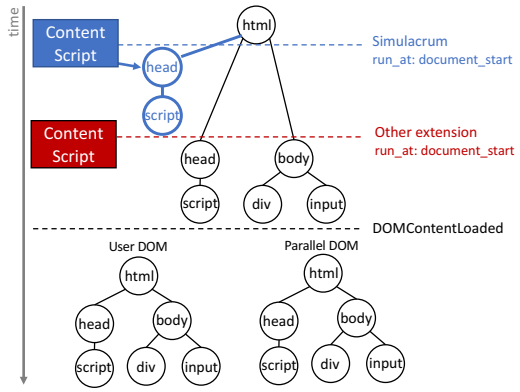

Figure 2: Different stages in a webpage's lifecycle.

execution context prior to DOM construction and before the browser executes any other script. Simulacrum achieves execution priority by setting `run_at: "document_start"` in the extension's manifest, which causes the browser to execute the extension's content scripts at the beginning of DOM construction and before any of the webpage's scripts. In this way, we guarantee that the webpage cannot recover the originals of the functions that we override by preemptively executing before Simulacrum's content scripts

However, when more than one extension sets `run_at: "document_start"`, the execution order is decided by the browser, based on the order of their installation time: the extension that was installed first will be executed before later-installed ones [44]. To ensure its prioritization, Simulacrum leverages the reordering technique proposed by Picazo-Sanchez et al. [44]. This approach uses the *management* permission to change the order of extensions in the execution pipeline by disabling and re-enabling them, resulting in these extensions being treated as if they had been installed after Simulacrum.

Figure 2 shows the different stages of loading a website when Simulacrum is present. The content script of Simulacrum runs at `document_start`. At this point, the browser has not started to parse the HTML page or construct the DOM. Simulacrum's content script creates a `head` element and appends its script to it. This script overrides the interfaces mentioned previously and then removes the `head` element from the page to prevent any conflict with the website's head element. Simulacrum reorders the execution pipeline of content scripts and will be the first script that executes in the webpage. Moving to the next area of the diagram in Figure 2, the browser starts to construct the DOM. Right after constructing the DOM, it fires the `DOMContentLoaded` and Simulacrum clones the DOM to generate the Parallel DOM. From this point forward, extensions' fingerprints remain on the User DOM and they will not be accessible by website scripts. Simulacrum's event listeners also fire before the webpage's and the other extensions. JavaScript orders the firing of event handlers based on when the listener was set. That is, even if multiple scripts set a listener for the `DOMContentLoaded` event, the browser invokes the Simulacrum's callback first (creating the Parallel DOM) because it is the first script that set a listener for the `DOMContentLoaded` event. In addition, JavaScript executes as a single thread; therefore, Simulacrum's event handler blocks other events from firing until Simulacrum's event handler completes (i.e., there is no race condition).

iframes. Simulacrum uses the `"all_frames": true` manifest file setting to inject its content scripts into all `iframes` so that it can protect against fingerprinting by code running in these `iframes`. However, due to browser implementation subtleties, `iframes` with a blank or undefined `src` attribute are not covered by the `all_frames` setting. Thus, an attacker can append `<iframe id="ifrm"></iframe>` to the DOM, retrieve the *original* version of `getElementById` through the `ifrm.contentWindow.Document.prototype.getElementById` lookup chain, and bypass Simulacrum. To inject Simulacrum's content scripts into such `iframes` (as well as `iframes` with src attributes of `javascript:*` or `about:*`), Simulacrum includes `"match_about_blank": true` in its manifest. This setting successfully injects the content scripts into all `iframes`. Interestingly, this bypass vector has been overlooked by some previously proposed countermeasures (e.g., [32]).

Another `iframe`-related circumvention can occur when fingerprinting code running in the parent page of an `iframe` exploits a race condition between the time points when an `iframe` is created and when Simulacrum's scripts override its DOM. The resulting attack is identical to the previously-stated lookup chain, depending on quick access to the original elements. Though this was reported in 2017 [50], it was never fixed, and popular privacy extensions (e.g., Privacy Badger [3]) are vulnerable to this attack. We prevent this attack by blocking all accesses to the `iframe`'s `contentWindow` and `contentDocument` from the parent page before Simulacrum has wrapped the `iframe`'s functions. To that end, the wrappers of `contentWindow` and `contentDocument` check Simulacrum's execution state inside the `iframe`; if it has completed, they return the function's result or `undefined` otherwise.

Overriding prototypes. The overriding approach from [47] would allow attackers to delete functions and revert them to the original function pointers. To prevent this and protect objects the authors used `Object.freeze` after the virtual machine layering process. In Simulacrum, as shown in Listing 1 (Appendix), we override prototypes and not objects. Our approach has two advantages; first, deleting functions will not revert them to the original ones. Second, since functions are not frozen, Simulacrum allows websites and extensions to wrap functions (i.e., they can add another layer on top of ours) which allows them to maintain their functionality.

5 Experimental Evaluation

Here we present a comprehensive evaluation that explores multiple dimensions of Simulacrum's performance.

Simulacrum's effectiveness. As our main goal is to mitigate browser extension fingerprinting that leverages DOM-based modifications, we need a robust and accurate fingerprinting system to evaluate our approach. To that end, we leverage the code and dataset of Carnus [27], as it is the only proposed system that *automatically* generates unique signatures that identify extensions based on their DOM interactions. While Carnus has modules for different types of extension techniques (e.g., detecting Web Accessible Resources) we focus on the DOM-based module. To account for the dynamic nature of extension behavior and to improve robustness, Carnus requires at least 90% of a fingerprint to match the DOM modifications to infer that an extension is installed. In their experiments, Karami et al. [27] reported 5,793 extensions being detected by Carnus based on their DOM-based fingerprints. To test the effectiveness of our proposed countermeasure, we replicate that experiment using the exact dataset of extensions. We use Selenium to open a fresh browser installed with Simulacrum. We then, using each of the 5,793 extensions installed, direct our browser to the website running Carnus' honeypage that will attempt to detect the extension. We find that Simulacrum is highly effective as it successfully prevents Carnus from detecting 5,553 (95.86%) of the extensions. In comparison, CloakX [61] could only protect up to 751 (12.96%) of these extensions [27] by randomizing a subset of node attributes.

Partial fingerprints. We further explore the effect of our defense and investigate extensions that Carnus is not able to detect, and find that certain extensions manage to inject a *subset* of their fingerprints in the Parallel DOM. This can occur due to various behaviors which we analyze over the following paragraphs, and Carnus does not detect them as it is not able to reach the 90% threshold required for identifying the extensions. Nonetheless we further assess the robustness of our defense by training Carnus on these partial fingerprints to examine whether it is able to identify any of these extensions.

To find extensions that partially inject fingerprints into the Parallel DOM, we modify Carnus to calculate the percentage of fingerprints that are injected into the Parallel DOM. This allowed us to identify 37 extensions with partial fingerprints. Our analysis reveals that the partial fingerprints of 28 extensions are unique enough within the entire collection of extensions fingerprints to identify those extensions. Therefore, a total of 268 extensions remain fingerprintable against Simulacrum, which amounts to only 4.63% of the extensions. Next we further explore the underlying reasons that lead to extensions not being protected by Simulacrum.

Document_start. We conducted an experiment on the 268 fingerprintable extensions to identify those that modify the DOM before the `DOMContentLoaded` event, which results in their fingerprints being included in the Parallel DOM. To that end, we installed each extension separately and visited the honeypage, taking a snapshot of the DOM at the `DOMContentLoaded` event, which we compared to the unmodified DOM. We find that 245 (4.22% overall) extensions make modifications before `DOMContentLoaded` as they are configured to run their content scripts at `document_start`, and alter the DOM before the construction of the Parallel DOM.

Next we evaluated the necessity of running the content script at `document_start` for extensions that modify the DOM before the `DOMContentLoaded` event. To that end, we randomly chose 20 extensions (that are in English and have at least 1k users) out of the 240 extensions, and modified the `run_at` attribute value in their `manifest` file from `document_start` to `document_idle`. We found that 19 extensions maintained their functionality despite our modification. Only one extension did not fully operate after the modification, which we believe was due to improper design patterns. Specifically, the developer of this extension implemented a non-critical logic at `DOMContentLoadedEvent`'s event handler and because we inject the code at `document_idle`, it does not work properly.

Injected scripts. While extensions have direct and full access to the DOM and are able to make all necessary DOM modifications through their content scripts, developers may decide to inject a `<script>` element into the page to make DOM modifications. To further explore this, we modify the 268 extensions to prevent them from running their content scripts at `document_start`. To achieve this, we replaced `document_start` with `document_idle` in the manifest file and background script. Second, we created another browser extension to identify `<script>` elements injected into the DOM. Third, we installed each modified extension besides Simulacrum and the new extension for collecting injected `<script>` elements and visited the Carnus honeypage; if we observe any injected `<script>` element we can conclude that it was injected by the extension. Finally, we identify if this injected `<script>` makes any DOM modifications. Since the injected `<script>` will not be transferred to the Parallel DOM, any modification in the Parallel DOM are the result of running this `<script>`. To find these modifications, we simply compare the Parallel DOM with its clean version. This way we found 40 (0.69% of the total) extensions that inject a `<script>` element into the DOM, and among these 17 extensions make modifications before the `DOMContentLoaded` event.

Effectiveness. Table 1 summarizes the Simulacrum's effectiveness. Overall, we find that it is highly effective at protecting extensions from DOM-based fingerprinting as it is able to protect 95.37% of the extensions that can be fingerprinted by Carnus. For the remaining 268 extensions that cannot be protected, we find that the majority is due to extension modifications being made at `document_start`. While our system cannot hide these extensions, we propose straightforward strategies for addressing the underlying issues in §6.

Simulacrum's impact. Next, we seek to measure the extent to which our defense affects the user experience. To that end we visit the top 1k websites and comparatively measure the overhead introduced by the presence of our extension, versus a browser without our extension. We focus on two key aspects: the page loading time and impact on functionality.

Table 1: Breakdown statistics for Simulacrum's effectiveness.

Extensions	5,793 (100%)
Protected	5,525 (95.37%)
Unprotected	268 (4.63%)
Document_start	245 (4.22%)
Script injection	40 (0.69%)
Partial fingerprints	28 (0.48%)

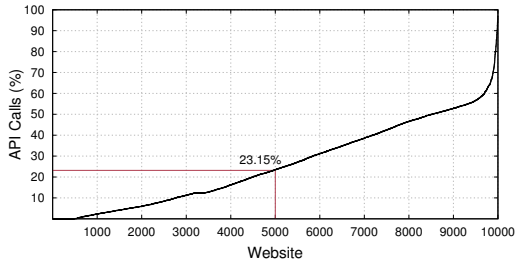

Figure 3: DOM-interacting APIs calls in the top 10K sites.

Function overriding. JavaScript has a set of functions for interacting with the DOM; by overriding them, Simulacrum adds overhead to their execution time. Here we investigate the top 10k websites to better understand the performance penalty introduced by our defense. To that end, we first use our VisibleJS extension to collect all the websites' API calls. In this experiment, we just collect the API calls that are executed after firing the `DOMContentLoaded` event, since only they incur Simulacrum's overhead. To identify which calls interact with the DOM, we assess the type of the arguments, receiver, and return values of all API calls. If they are a node or any other DOM-related object, we consider the API call to be interacting with the DOM. We exclude API calls that are used for managing data structures, such as functions that are implemented by the `Set`, `Array`, or `Map` interfaces. As an example, while the argument of `array.push(node)` is a node, we do not count it as a DOM interaction.

Figure 3 depicts the percentage of API calls that interact with the DOM in each of the top 10K websites. We find that for 85% of the sites, Simulacrum does not intercept half of the JavaScript API calls, while for half of the sites almost 76.85% of the calls are unaffected. We also plot the 20 most commonly invoked interfaces in Figure 4. The most common interface is `String`, used more than one billion times in the top 10K websites. Among these interfaces we only override the eight highlighted interfaces as Simulacrum does not need to modify the rest. We find that the invocations of `String` are 2.5x those of `Node` and 31x those of `Document`. Generally, we observe that the majority of JavaScript API calls do not interact with the DOM and are thus not affected by Simulacrum. Next, we focus on more precisely quantifying our performance overhead.

DOM loading time. In this experiment we visit the top 1k websites with two browser instances, one that has our extension installed and one without, and compare the time required for loading the websites' DOM. To account for uncontrollable fluctuations in the environment (network latency, OS scheduling, etc.), we average the results for each website over 10 runs. Overall, we were able to obtain both loading times for 925 out of the 1k websites that we visited. We observed that for 19 of the remaining 75 websites the page loaded successfully when visiting with a vanilla browser but did not complete loading when using a Simulacrum browser. The remaining 56 did not load successfully when using a vanilla browser but, interestingly, five of them loaded when visiting with Simulacrum installed.

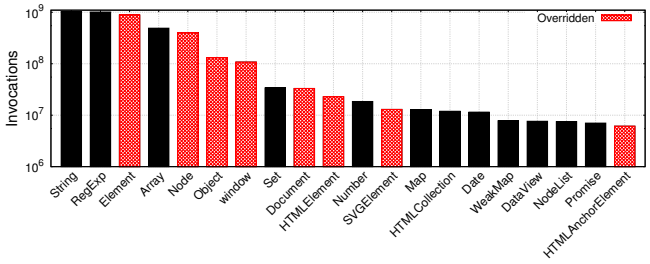

Figure 4: Interface invocations in the top 10K websites. Interfaces overridden by Simulacrum are explicitly noted.

In Figure 5 we present the difference in DOM loading times for the 925 websites in both absolute and relative measurements. We observe that 69% of those websites incur less than a one second overhead, demonstrating the feasibility and practicality of our approach. We also measured the delay imposed by overriding all relevant JavaScript functions, and the overhead of inserting noise; these amount to an average of 14.8ms and 31ms respectively.

Regarding DOM loading times, we observe a negative difference for 63 websites (6.81%) and a positive difference for 862 websites (93.19%). In the case of negative differences, these websites appear to complete loading faster when we visit them with a browser that has our extension installed. We found that half of these websites (33 out of 63) have a "speed-up" lower than 200 milliseconds, and that only 6 of the websites appear to surpass one second. While small, such negative differences can be attributed to network delays. In the case of significant differences this could be the result of our defense affecting a website's functionality or blocking the retrieval of content, ads, etc., and thus complete loading faster. For the 862 websites that have a positive difference, where our defense introduces a delay overhead, we found that this delay is 895 milliseconds (116% relative) on average; 563 of the websites exhibit more than 50% relative overhead, with 254 sites between 50%-100%, 194 between 100%-200%, and 115 sites with more than 200% of overhead, which heavily skews the overall overhead, as can be seen in Figure 5. More specifically, we observe that the delay for half of the websites (431 out of 862) is below 390 milliseconds (38% relative) and that only 11.72% (101 out of 862) observe delays that exceed two seconds (224% relative). When

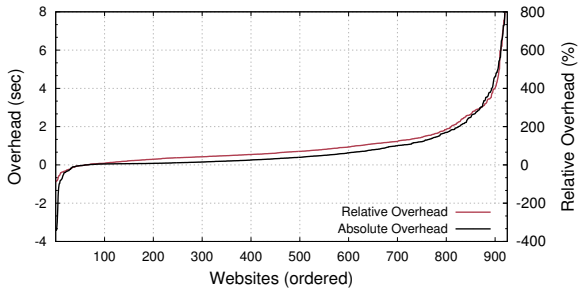

Figure 5: DOM loading time difference caused by Simulacrum.

extensions that remove page elements are present, Simulacrum activates an additional protection for replacing these elements (see position-based attacks, §4.4), and an additional overhead of 172 milliseconds is incurred per website on average.

User experience. As the additional JavaScript code and DOM manipulations might impact the user experience, we experimentally quantify the impact using Google Lighthouse [23]. Lighthouse is an open-source automated tool for obtaining performance metrics and insights [23], which provides an overall performance score based on six metrics; among those, we are specifically interested in the `time-to-interactive (TTI)` metric as it is not affected by network fluctuations. The TTI metric focuses on the amount of time it takes for a web page to become fully interactive. To measure TTI, Lighthouse waits for the page to display useful content and registers event handlers for most visible page elements.

To evaluate Simulacrum on sites with heavy JavaScript usage we refer to BuiltWith [10], which publishes a list of popular JavaScript technologies across the web. We select five popular technologies and for each we randomly choose ten out of the list of top 50 websites. We run the Google Lighthouse performance analysis tool against each website, with and without Simulacrum installed, and visit each website ten times in each setup. We calculate the average `TTI` overhead for websites that loaded and had a positive overhead. As can be seen in Table 2, in most cases the overhead is less than one second, while the largest overhead of 2.47 seconds is observed in sites built with `Bootstrap`. When taking into consideration the average page loading times recently reported for popular websites in the US [25] (with ~60% and 18% of pages requiring more than 10 and 30 seconds respectively) we argue that the overhead introduced by our prototype is reasonable given the privacy enhancements offered by our approach. We believe that this tradeoff can be significantly improved by leveraging DOM Reality Shifting for deploying defenses against other classes of web attacks, which we consider as part of our future work.

Furthermore, we evaluated Simulacrum using the *Celtic Kane* [6] open-source benchmark for JavaScript and DOM speed tests. The results show that Simulacrum does not influence the duration of most test cases, as shown in Figure 7 in the Appendix. While the DOM benchmark increases from 8ms to 38ms, overall our performance tests indicate that overhead is negligible or within acceptable ranges when compared to the loading times and responsiveness of modern websites.

Table 2: Average Time-To-Interactive overhead in seconds.

Technology	TTI Overhead
core-js	1.44 (17.5%)
facebook-SDK	0.99 (19.35%)
react	0.65 (16.56%)
bootstrap	2.47 (24.76%)
lodash	0.58 (16.53%)

Memory overhead. We crawled the top 1k websites with and without Simulacrum and measured the heap usage of JavaScript (`usedjsHeapSize/jsHeapSizeLimit`) using the `performance.memory` API (this experiment does not measure memory consumption for same-site iframes and workers and also ignores garbage collection). We visited websites five times and found an average increase in consumption of only 0.25%.

Website breakage. Next we explore how our anti-fingerprinting extension impacts the general functionality of web applications. To evaluate websites' functionality, we randomly selected 50 websites among the top 100 Alexa list and interacted with each website by testing common operations with Simulacrum installed (we provide more details on our approach in the Appendix). We opted for popular websites as they tend to offer rich functionality and make heavy use of a wide range of JavaScript features, and their complexity allows us to stress test our prototype. We also note that we followed a *continuous testing* approach while developing Simulacrum, which allowed us to identify implementation peculiarities in websites and JavaScript frameworks/libraries, which allowed us to refine our code for handling such cases. For instance, we initially faced some issues in Instagram that were the result of the shallow JavaScript node cloning API, which in turn led us to improve our implementation of the `deepClone` function. Assigning an `id` attribute to all page elements was also challenging for a website such as Spotify; we handled it by keeping a record of elements that do not require an `id` attribute and tuning the corresponding `getter` functions.

Table 3 details the breakage caused by our system, with major breakage in 6 sites (12%). Submitting credentials in the Google and Amazon login pages results in a crash or an error message. LinkedIn successfully passes authentication and displays contents of the main page, however navigating through the website is not smooth and contents might not show properly. Finally, product details on the main Ebay page are missing, but we can look up items in the website's search bar. We also found 5 websites (10%) with minor issues that do not drastically impact websites' core functionality, such as affecting the page's appearance or certain menu options being disabled. While our system does not affect the majority of websites that we tested, privacy-preserving defenses typically introduce a trade-off between privacy and functionality.

Table 3: Breakage in 50 randomly-chosen popular websites.

Breakage	Statistics	Websites
Major	6 (12%)	
Authentication	2 (4%)	google.com, amazon.com
Missing Content	2 (4%)	linkedin.com, ebay.com
Disabled Actions	2 (4%)	twitch.tv, etsy.com
Minor	5 (10%)	
Disabled Elements	1 (2%)	csdn.net
Appearance	2 (4%)	tmall.com, office.com
Other	2 (4%)	instagram.com, apple.com

Finally, Simulacrum supports allowlisting websites.

Extension breakage. While our experiments demonstrate that Simulacrum is highly effective in hiding the presence of installed extensions, we also need to verify that this protection does not hinder extensions' functionality. Therefore, we randomly chose 50 English extensions from the list of protected extensions to evaluate their functionality. For each extension, we manually explore the extension to learn its functionality in a clean Chrome browser, and then extensively re-evaluate its functionality in the presence of Simulacrum. Our evaluation shows that all of the extensions work properly and retain their complete functionality, thus demonstrating that our system can effectively protect the randomly chosen extensions without hindering their functionality. We also experimentally verified that Simulacrum does not interfere with other privacy extensions that wrap APIs, by testing it with Privacy Badger [3], AdBlock [1] and uBlock [5] across five popular news websites.

6 Discussion, Limitations, and Guidelines

Browser-based defense. While Simulacrum effectively addresses the root cause of behavior-based fingerprinting, our experimental evaluation reveals certain corner cases or uncommon extension behaviors that lend themselves to fingerprinting. However, these idiosyncrasies are not fundamental to the extension's actual functionality and may even be the result of developer oversight. As such, we provide explicit guidelines for developers that allow them to protect their extensions from being fingerprinted through minimal changes that do not affect the extension's functionality. We hope that our work inspires browser vendors to consider incorporating our defensive mechanism as native browser functionality, as it would allow the browser to address cases that cannot be handled by our solution due to the inherent limitations of an extension-based defense, while further reducing overhead. We note that this is theoretically feasible but might, nevertheless, require significant effort or changes to a browser's design.

Extension modification timing. For changes that occur at `document_start`, modifications can be postponed to the `DOMContentloaded` event without affecting the extensions' functionality or the user experience in the vast majority of cases. Unless this is absolutely necessary for functionality, developers should opt for running at `document_idle` or wait for the `DOMContentloaded` event to be fired prior to making changes.

Injected scripts. We found 40 fingerprintable extensions injecting scripts that modify the DOM instead of directly modifying it from their content script. As the modifications occur from a script being executed in the page's execution environment, Simulacrum does not exclude them from the Parallel DOM. However, this approach does not offer any additional capabilities, and we urge extension developers to limit DOM-changing behavior to content scripts.

Breakage. Debugging popular websites and identifying the root causes of breakage requires significant engineering effort due to the use of advanced JavaScript libraries, obfuscation, and code minification. The most common cause of breakage early on was the *shallow clone* problem (see §4.1), which introduced inconsistencies between the Parallel and User DOM. Since Simulacrum is a research prototype that highlights the efficacy of DOM Reality Shifting for preventing DOM-based extension fingerprinting, we believe that our system's effects on JavaScript-heavy websites is acceptable, and should be incorporated by browser vendors into the browser. That engineering process would also likely address all breakage.

Simulacrum detectability. In practice, websites can infer the presence of our extension (e.g., by observing the `id` attributes that are added to elements). However, this leaks minimal entropy compared to the entropy obtained by fingerprinting multiple installed extensions (which can be uniquely identifying [24, 27]), since the leaked information is a binary attribute representing whether Simulacrum is present (e.g., note the entropy of binary attributes compared to more complex ones like the `User-Agent` or the list of fonts [31]). Since the anonymity offered by privacy-enhancing technologies (PETs) correlates with the number of users [18], wide adoption of our defense would further decrease the entropy leak (similar to any other PET, e.g., Tor [4]). If our defense was incorporated into the browser this leak would be eliminated. Moreover, Simulacrum has the additional privacy benefit of preventing the inference attacks presented in [27].

Timing attacks. An interesting class of attacks are side-channel timing attacks. This would involve page JavaScript inferring if an extension added or modified an element due to Simulacrum executing different code paths. Similarly, Goethem and Joosen [62] exploited timing differences in evaluating the `web_accessible_resources` property inside an extension's `manifest` file to infer its existence. In general timing attacks are fairly tricky to implement, detect, and prevent, yet history has shown that they only increase in effectiveness. While such attacks are outside our current threat model, they represent an exciting avenue for future work.

Manual analysis. Simulacrum uses different wrappers for functions that are assigned to different categories. Categorizing functions relies on analyzing their behavior, which requires assigning the correct argument(s) and receiver, and observing the resulting behavior (which is not necessarily visible through

JavaScript). Therefore, we manually analyze functions to ensure we override them correctly. It is also important to note that JavaScript interfaces tend to remain stable, and any potential changes to methods can be easily handled. Moreover, any new DOM APIs that appear can be readily integrated following our existing implementation templates.

7 Related Work

Numerous papers have demonstrated or leveraged browser extension fingerprinting techniques [24, 27, 46, 49, 57, 62]. Several studies over the past few years have either implemented or proposed a wide range of anti-fingerprinting techniques; however, all of the methods differ in scope or capability from the complete DOM-based-fingerprinting prevention implemented by Simulacrum. As prior defenses mostly focus on other fingerprinting techniques, those studies are complementary to ours and, in practice, could be combined into a more holistic and comprehensive anti-fingerprinting defense.

Prior work has enabled users to manually disable extensions on specific websites [48, 55]. Simulacrum differs as it works automatically without requiring user interventions or decision-making. Other work [56] hides user interests by randomly visiting websites, defending against a complementary fingerprinting technique. Sanchez-Rola et al. [46] devised a timing-based method for fingerprinting extensions and proposed modifications to the web browser's extension invocation to prevent the timing-based attack.

Most related to Simulacrum is CloakX [61], a defense that modifies publicly accessible extension identifiers to prevent fingerprinting. The modifications randomize web-accessible resources [24, 49] as well as other identifiers. However, CloakX does not address behavioral fingerprinting and does not prevent the majority of fingerprints generated by Carnus [27]. Conversely, Simulacrum effectively targets the root cause of the attack, while employing a less invasive approach that does not require modification of the user's extensions.

Recent work proposed a countermeasure against CSS-based extension fingerprinting [32]. Their defense creates a mirror copy of the DOM tree using Shadow DOM, which automatically excludes content styles from extensions. The defense overrides the `getComputedStyle` method of each element to return the style of the Shadow DOM elements so as not to include the content styles injected by extensions. While this work protects against a different class of fingerprinting than Simulacrum, its proposed CSS defense would benefit from using our techniques. It is important to note that the `Shadow DOM` API [37] is not sufficient for implementing our defense, as it was not designed to secure or hide page elements; instead, it encapsulates elements to prevent problems such as variable name collisions. Using it does not prevent a malicious script from accessing the `Shadow DOM`'s content (even when attached in `closed` mode). Therefore, the `Shadow DOM` does not satisfy our security requirements. Unlike any prior work, Simulacrum robustly overrides 1,532 functions that isolate DOM modifications and prevents DOM-based fingerprinting.

Simulacrum extends the concept of `virtual machine layering`, which creates an abstraction layer on top of the JavaScript VM that controls sites' access to APIs. This was used for instrumenting JavaScript [33], enabling replay [38], and controlling access to APIs [47, 51]. Photon [33] transforms JavaScript (outside the browser) to create a VM layer that developers can use to instrument and evaluate their code. Mugshot [38] captures and replays JS events using a server-side web proxy. This only requires a single overriding strategy, removing the need for categorizing functions (as we do for Simulacrum) and allowing for straightforward automated overriding; this, however, would not be sufficient for the challenging process necessitated by our defense. Moreover, unlike Photon and Mugshot, anyone can use Simulacrum to protect their privacy *without* requiring any external operations (e.g., translation or a server-side web proxy) or user intervention. Even though Chrome Zero [47] and the tool from [51] enhance privacy by allowing users to limit sites' access to JavaScript APIs, their approach is inherently designed to *prevent website functionality* while our system focuses on *maintaining* it.

8 Conclusions

As privacy-invasive tactics remain rampant on the web, developing privacy-enhancing countermeasures that augment the protections currently offered by browsers is of paramount importance. We proposed DOM Reality Shifting as a strategy for tackling a particularly robust extension-fingerprinting method that identifies how extensions' functionality modifies the web page. The inner workings and implementation details of our prototype extension highlight the technical challenges of implementing our strategy in practice, due to the inherent complexities and peculiarities of the web. Nonetheless, our evaluation demonstrates Simulacrum's effectiveness against the state-of-the-art DOM-based fingerprinting system, while introducing minimal performance overhead. Overall, we envision DOM Reality Shifting as a building-block for various advanced privacy-enhancing browser countermeasures.

Acknowledgments

We would like to thank the anonymous reviewers, and our shepherd Roberto Perdisci, for their valuable feedback that helped us improve our system. This work was supported by the Office of Naval Research (ONR) under grant N00014-21-1-2159, the Defense Advanced Research Projects Agency (DARPA) under Grant No. N66001-20-C-4020, and the National Science Foundation (NSF) under grants CNS-1934597, CNS-1703644 and CNS-1651661. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government.

References

[1] Adblock. https://chrome.google.com/webstore/detail/adblock-%E2%80%94-best-ad-blocker/gighmmpiobklfepjocnamgkkbiglidom.

[2] EasyList - Overview. https://easylist.to/.

[3] Privacy badger. https://chrome.google.com/webstore/detail/privacy-badger/pkehgijcmpdhfbdbbnkijodmdjhbjlgp.

[4] Tor project | anonymity online. https://www.torproject.org.

[5] ublock origin. https://chrome.google.com/webstore/detail/ublock-origin/cjpalhdlnbpafiamejdnhcphjbkeiagm.

[6] Web browser javascript benchmark. http://celtickane.com/labs/web-browser-javascript-benchmark.

[7] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *CCS*, 2014.

[8] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. Fpdetective: dusting the web for fingerprinters. In *CCS*, 2013.

[9] Assel Aliyeva and Manuel Egele. Oversharing is not caring: How cname cloaking can expose your session cookies. In *Asia CCS*. Association for Computing Machinery, 2021.

[10] BuiltWith. Javascript usage distribution in the top 1 million sites. https://trends.builtwith.com/javascript.

[11] Yinzhi Cao, Song Li, and Erik Wijmans. (cross-)browser fingerprinting via OS and hardware level features. In *NDSS*, 2017.

[12] Phakpoom Chinprutthiwong, Raj Vardhan, GuangLiang Yang, Yangyong Zhang, and Guofei Gu. The service worker hiding in your browser: The next web attack target? In *RAID*, 2021.

[13] World Wide Web Consortium. DOM Living Standard. https://dom.spec.whatwg.org/.

[14] World Wide Web Consortium. Intersection Observer. https://w3c.github.io/IntersectionObserver/.

[15] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. The web's sixth sense: A study of scripts accessing smartphone sensors. In *CCS*, 2018.

[16] Louis F DeKoven, Stefan Savage, Geoffrey M Voelker, and Nektarios Leontiadis. Malicious browser extensions at scale: Bridging the observability gap between web site and browser. In *10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)*, 2017.

[17] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom Van Goethem. The cname of the game: Large-scale analysis of dns-based tracking evasion. *PETS*, 2021.

[18] Roger Dingledine and Nick Mathewson. Anonymity loves company: Usability and the network effect. In *WEIS*, 2006.

[19] Peter Eckersley. How unique is your web browser? In *PETS*, 2010.

[20] Steven Englehardt and Arthur Edelstein. Firefox 85 cracks down on supercookies. https://blog.mozilla.org/security/2021/01/26/supercookie-protections/, 2021.

[21] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *CCS*, 2016.

[22] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In *WWW*, 2018.

[23] Google. Lighthouse. https://developers.google.com/web/tools/lighthouse.

[24] Gabor Gyorgy Gulyas, Doliere Francis Some, Nataliia Bielova, and Claude Castelluccia. To extend or not to extend: on the uniqueness of browser extensions and web logins. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*, 2018.

[25] Molly Hanson, Patrick Lawler, and Sam Macbeth. The tracker tax: the impact of third-party trackers on website speed in the united states. Technical report, 2018.

[26] Soroush Karami, Panagiotis Ilia, and Jason Polakis. Awakening the web's sleeper agents: Misusing service workers for privacy leakage. In *NDSS*, 2021.

[27] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. Carnus: Exploring the privacy threats of browser extension fingerprinting. In *NDSS*, 2020.

[28] Amit Klein and Benny Pinkas. Dns cache-based user tracking. In *NDSS*, 2019.

[29] Brian Kondracki, Assel Aliyeva, Manuel Egele, Jason Polakis, and Nick Nikiforakis. Meddling middlemen: Empirical analysis of the risks of data-saving mobile

browsers. In *IEEE Symposium on Security and Privacy*, 2020.

[30] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)*, 2020.

[31] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE Symposium on Security and Privacy*, 2016.

[32] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. Fingerprinting in style: Detecting browser extensions via injected style sheets. In *USENIX Security*, 2021.

[33] Erick Lavoie, Bruno Dufour, and Marc Feeley. Portable and efficient run-time monitoring of javascript applications using virtual machine layering. In *European Conference on Object-Oriented Programming*, 2014.

[34] Xu Lin, Panagiotis Ilia, and Jason Polakis. Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill. In *CCS*, 2020.

[35] MDN. MDN Object Model. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model.

[36] MDN. MDN Web API. https://developer.mozilla.org/en-US/docs/Web/API.

[37] MDN. Mdn web docs - using shadow dom. https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM.

[38] James W Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, 2010.

[39] Vikas Mishra, Pierre Laperdrix, Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Martin Lopatka. Don't count me out: On the relevance of ip address in the tracking ecosystem. In *Proceedings of The Web Conference*, 2020.

[40] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In *Proceedings of W2SP*, 2012.

[41] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, Edgar Weippl, and FC Wien. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, 2013.

[42] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Symposium on Security and Privacy*, 2013.

[43] Raffaello Perrotta and Feng Hao. Botnet in the browser: Understanding threats caused by malicious browser extensions. *IEEE security & Privacy*, 2018.

[44] Pablo Picazo-Sanchez, Juan Tapiador, and Gerardo Schneider. After you, please: browser extensions order attacks and countermeasures. *International Journal of Information Security*, 2019.

[45] Corey Prophitt. Nefarious linkedin. https://github.com/dandrews/nefarious-linkedin, 2017.

[46] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In *USENIX Security*, 2017.

[47] Michael Schwarz, Moritz Lipp, and Daniel Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS*, 2018.

[48] Alexander Sjösten, Steven Van Acker, Pablo Picazo-Sanchez, and Andrei Sabelfeld. Latex gloves: Protecting browser extensions from probing and revelation attacks. In *NDSS*, 2019.

[49] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. Discovering browser extensions via web accessible resources. In *CODASPY*, 2017.

[50] Peter Snyder. Issue 793217: "document_start" hook on child frames should fire before control is returned to the parent frame. https://bugs.chromium.org/p/chromium/issues/detail?id=793217.

[51] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. Browser feature usage on the modern web. In *IMC*, 2016.

[52] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *CCS*, 2017.

[53] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. Tales of favicons and caches: Persistent tracking in modern browsers. In *NDSS*, 2021.

[54] Dolière Francis Somé. Empoweb: Empowering web applications with browser extensions. In *IEEE Symposium on Security and Privacy*, 2019.

[55] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *WWW*, 2019.

[56] Oleksii Starov and Nick Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *WWW*, 2017.

[57] Oleksii Starov and Nick Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *IEEE Symposium on Security and Privacy*, 2017.

[58] Karthika Subramani, Xingzi Yuan, Omid Setayeshfar, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. When push comes to ads: Measuring the rise of (malicious) push advertising. In *IMC*, 2020.

[59] Paul Syverson and Matthew Traudt. Hsts supports targeted surveillance. In *8th USENIX Workshop on Free and Open Communications on the Internet (FOCI '18)*, 2018.

[60] David Temkin. Google Ads - Charting a course towards a more privacy-first web. https://blog.google/products/ads-commerce/a-more-privacy-first-web/, 2021.

[61] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. Everyone is different: Client-side diversification for defending against extension fingerprinting. In *USENIX Security*, 2019.

[62] Tom Van Goethem and Wouter Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In *WOOT*, 2017.

[63] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-stalker: Tracking browser fingerprint evolutions. In *IEEE Symposium on Security and Privacy*, 2018.

[64] Zachary Weinberg, Eric Y Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *IEEE Symposium on Security and Privacy*, 2011.

[65] John Wilander. Webkit - intelligent tracking prevention (itp). https://webkit.org/blog/9521/intelligent-tracking-prevention-2-3/, 2019.

[66] CSS Working Group World Wide Web Consortium. Resize Observer. https://drafts.csswg.org/resize-observer/.

Appendix

Overriding prototype methods. We employ the following steps: keeping a reference to the original function (Line 1); declaring and defining the new wrapper function (Lines 2-7); implementing the overriding logic (Lines 3, 5); executing the original function on proper inputs (Line 4); returning the desired value from the function (Line 6); and assigning the wrapper function to the object's prototype to overwrite the original method (Line 8).

```
1 let originalFunction = Parent.prototype.function;
2 let wrapperFunction = function() {
3   // [...] Custom logic
4   let returnValue = originalFunction.apply(obj, args);
5   // [...] Custom logic
6   return returnValue
7 }
8 Parent.prototype.function = wrapperFunction;
```

Listing 1: Overriding `function` from `Parent`'s prototype.

Overriding object properties. Listing 2 shows how we use `Object.defineProperty()` to override the `setter` and `getter` functions for the `id` property of the `parentObj` prototype.

```
1  function overrideProperty(){
2    let propDes =
3    ↳ Object.getOwnPropertyDescriptor(parentObj.prototype,'id');
4    let getRef = propDes.get;
5    let setRef = propDes.set;
6    Object.defineProperty(parentObj.prototype,'id', {
7      get: function() {
8        let retValue = getRef.apply(obj, args);
9        // Overriding logic
10       return retValue;
11     },
12     set: function() {
13       let retValue = setRef.apply(obj, args);
14       // Overriding logic
15       return retValue;
16     },
17     configurable: true,
18     enumerable: true
19 });}
```

Listing 2: Overriding `id` property from `parentObj`'s prototype; `obj` and `args` values change based on our logic.

Recording function invocations. We created the extension VisibleJS to gather information related to DOM usage by websites. Listing 3 shows the script VisibleJS uses to wrap all the JavaScript interfaces. While the webpage runs VisibleJS reports the interfaces used by the website.

```
1 let orig = original[<interface.method>];
2 interface.prototype['method'] = function(){
3    let toReturn = orig.apply(this, arguments);
4    collect(this, arguments, toReturn);
5    return toReturn;
6 }
```

Listing 3: VisibleJS general strategy.

Observer interfaces. Listing 4 shows simple examples that demonstrate using three different types of observers. `ResizeObserver` does not need a configuration parameter. `IntersectionObserver` receives the configuration and callback parameters when it is constructed. `MutationObserver` receives the callback during construction and receives the target and the configuration details when the observe method is invoked.

```
// Resize Observer
observer = new ResizeObserver(callback)
observer.observe(target)

// Intersection Observer
configuration = {root: element}
observer = new IntersectionObserver(callback, configuration)
observer.observe(target)

// Mutation Observer
configuration = {childList: true, subtree: true}
observer = new MutationObserver(callback)
observer.observe(target, configuration)
```

Listing 4: Observer instantiation examples.

MutationObserver. Listing 5 shows the wrapper code used in the `MutationObserver`'s constructor. Simulacrum overrides `MutationObserver`'s constructor to modify the incoming callback. Before calling the original observer method, the wrapper filters out the mutations that originate from an extension.

```
original = MutationObserver;
class newMutationObserver{
  constructor(){
    callback = arguments[0];
    newCallback = function(){
      mutations = filterMutations(arguments[0])
      if (mutations.length>0)
        callback(mutations);
    }
    return new original(newCallback);
}}
MutationObserver = newMutationObserver;
```

Listing 5: Wrapping the constructor of `MutationObserver`.

Overriding functions. Table 4 shows the number of overridden functions. The first column lists the noteworthy interfaces. The leftmost grouping breaks the interfaces down by properties and methods overridden. On the right side, the table breaks the interfaces down by the type of wrapper used, with the most common category being the simple setter and getter categories.

Examples of overridden functions. Table 5 displays different function examples for each category with their overridden counterpart. For readability reasons we omit details in the "Overridden Function" column. For example, we use the "original" hash table (§4.1) for calling the functions but we do not include them in this table for simplicity. That is, instead of `original["Element.innerHTML_setter"].call(e,"text")`, we write `e.innerHTML="text"`.

Inheritance hierarchy. In Figure 6 we provide a partial

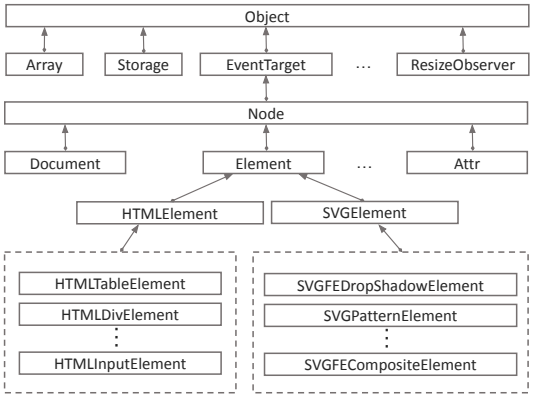

Figure 6: Inheritance hierarchy in the DOM API.

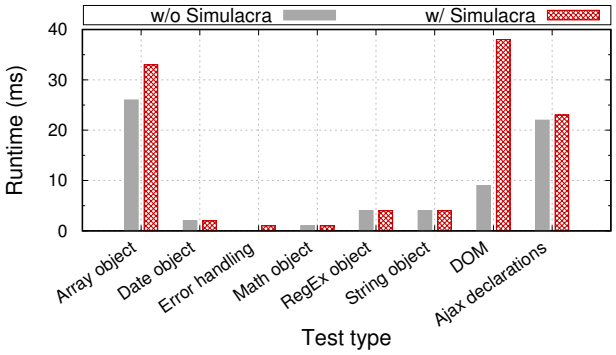

Figure 7: Run-time overhead when evaluating Simulacrum with the Celtic Kane open-source benchmark speed tests.

representation of the inheritance hierarchy in respect to the DOM API and some of the interfaces pertinent to our work.

Breakage tests. We manually assessed Simulacrum's effect on extension and website functionality by testing common operations with Simulacrum installed. We had a user freely explore websites with and without Simulacrum installed for a few minutes and then report any effects to the browsing experience when Simulacrum was present. This included assessing each website's functionality (e.g., login and payment processes, playing videos, interacting with website-specific tools, etc.) and was not limited to a visual inspection. For the scenarios testing the impact on other extensions, a user first installed the candidate extension and learned their expected functionality on a few target websites (either the ones advertised in the extension's overview page in the Chrome web store or on websites of user's choice). Subsequently, the expected functionality was reevaluated with Simulacrum installed. In the case of privacy extensions (Privacy Badger [3], AdBlock [1] and uBlock [5]) we specifically chose five popular news websites (`nytimes.com`, `foxnews.com`, `abcnews.com`, `cnn.com` and `bbc.com`) which usually involve tracking scripts and verified that these privacy extensions continued to operate as expected and their functionality was not affected by Simulacrum.

Table 4: The number of overridden functions for noteworthy interfaces.

Interface	Properties	Methods	Total	Overridden Functions				
				Simple Setter	Simple Getter	Active Setter	Active Getter	Arguments
Node	14	15	29	7	23	0	1	8
Document	264	40	304	20	34	115	153	6
Element*	1,283	189	1,472	687	968	84	193	14

* Element interface and all other interfaces that inherit from it.
Note: Some properties have both a setter and getter, thus the sum of used strategies is more than the total.

Table 5: Examples of different functions and our strategy for wrapping them.

Category	Example	Overridden Function
Simple getter	`document.getElementsByTagName("div")`	`parallelDOM.getElementsByTagName("div")`
Simple getter	`e.hasAttribute("src");`	`parallelDOM.getElementById(e.id).hasAttribute('src')`
Active getter	`e.scrollTop`	`if (parallelDOM.getElementById(e.id))` `  userDOM.getElementById(e.id).scrollTop`
Active getter	`document.activeElement`	`result = userDOM.activeElement;` `return parallelDOM.getElementById(result.id);`

Simple setter	`e.innerHTML = "text"`	`equivalentElement = getEquivalent(e);`
		`e.innerHTML = "text";`
		`equivalentElement.innerHTML = "text";`

Simple setter	`document.write("text")`	`document.write("text")` `parallelDOM.write("text");`
Active setter	`e.requestFullscreen()`	`if (parallelDOM.getElementById(e.id))` `  userDOM.getElementById(e.id).requestFullscreen()`

Arguments	`n.insertBefore(newNode, refNode)`	`equivNewNode = deepClone(newNode);`
		`equivRefNode = getEquivalent(refNode);`
		`equivNode = getEquivalent(n);`
		`n.insertBefore(newNode, refNode);`
		`equivNode.insertBefore(equivNewNode, equivRefNode);`

Benchmarking tests. In Figure 7 we present the results from our performance test using the Celtic Kane open-source benchmark. We find that in most cases Simulacrum does not affect performance, while for the DOM test our system introduces a 30 millisecond overhead.

Noise-based defense against position-based attacks. In Figure 8 we provide an example screenshot that shows a page's visual appearance when Simulacrum introduces noisy elements as a countermeasure against position-based attacks. These elements are simply empty spaces that change the position of other elements on the page.

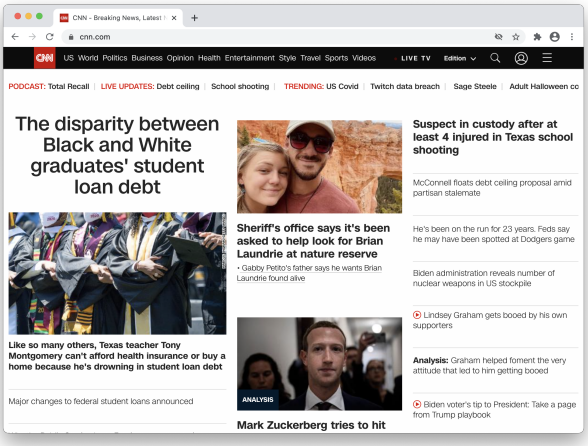

Figure 8: Example of site with added noise elements.