DeView: Confining Progressive Web Applications by Debloating Web APIs

ChangSeok Oh
Georgia Institute of Technology
Atlanta, United States
changseok@gatech.edu

Sangho Lee
Microsoft Research
Redmond, United States
Sangho.Lee@microsoft.com

Chenxiong Qian
University of Hong Kong
Hong Kong, China
cqian@cs.hku.hk

Hyungjoon Koo*
Sungkyunkwan University
Suwon, South Korea
kevin.koo@skku.edu

Wenke Lee
Georgia Institute of Technology
Atlanta, United States
wenke@cc.gatech.edu

ABSTRACT

A progressive web application (PWA) becomes an attractive option for building universal applications based on feature-rich web Application Programming Interfaces (APIs). While flexible, such vast APIs inevitably bring a significant increase in an API attack surface, which commonly corresponds to a functionality that is neither needed nor wanted by the application. A promising approach to reduce the API attack surface is software debloating, a technique wherein an unused functionality is programmatically removed from an application. Unfortunately, debloating PWAs is challenging, given the monolithic design and non-deterministic execution of a modern web browser. In this paper, we present DeView, a practical approach that reduces the attack surface of a PWA by *blocking* unnecessary but accessible web APIs. DeView tackles the challenges of PWA debloating by i) *record-and-replay web API profiling* that identifies needed web APIs on an app-by-app basis by replaying (recorded) browser interactions and ii) *compiler-assisted browser debloating* that eliminates the entry functions of corresponding web APIs from the mapping between web API and its entry point in a binary. Our evaluation shows the effectiveness and practicality of DeView. DeView successfully eliminates 91.8% of accessible web APIs while i) maintaining original functionalities and ii) preventing 76.3% of known exploits on average.

CCS CONCEPTS

• **Security and privacy → Browser security**; **Web application security**.

KEYWORDS

Debloating; Browser; Program Analysis; Record-and-Replay; PWA; Progressive Web Application; Web APIs

*Corresponding author

ACSAC '22, December 5–9, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9759-9/22/12.
https://doi.org/10.1145/3564625.3567987

ACM Reference Format:
ChangSeok Oh, Sangho Lee, Chenxiong Qian, Hyungjoon Koo, and Wenke Lee. 2022. DeView: Confining Progressive Web Applications by Debloating Web APIs . In *Annual Computer Security Applications Conference (ACSAC '22), December 5–9, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3564625.3567987

1 INTRODUCTION

Building a web application to provide a service becomes an attractive option for both desktop and mobile developers [89, 93] because web Application Programming Interfaces (APIs) can offer nearly complete features while being independent of a platform. Furthermore, unlike a conventional software life cycle that entails rebuilding and redistributing an application (e.g., via app-store) for different platforms, web application development enables quick updates and deployments because a web browser transparently fetches and runs updated web applications.

Aside from such advantages, web applications have a few limitations, such as the absence of ① access to underlying system services and hardware, ② persistent client-side storage for offline operations, and ③ application meta-data (e.g., name, description, and version). In response, a hybrid approach has been introduced to complement the above shortcomings by allowing a native application to harness web APIs via an embedded system service (e.g., WebView [27, 46, 58, 65, 70]) or a standalone framework (e.g., Electron [41]). However, such approaches provide additional functionality at the expense of platform independence because its native portion is still tightly coupled with an underlying platform. Moreover, building a desktop application with Electron [41] maintains its copies by design.

A progressive web application (PWA) [92] is a recent effort to build native-like applications with web technologies. The core enabler of a PWA is the HTML5 features [122] that ① access system services and hardware (e.g., `WebAudio` [1], `WebRTC` [61], `WebUSB` [47]), ② hold a resource offline within persistent browser-side storage (e.g., `Storage` [3], `Service Worker` [102]), and ③ describe application information separately (e.g., `Web Application Manifest` [36]). In this respect, PWAs gained in popularity on both desktop and mobile platforms [94], expecting to surpass 10 billion USD by 2027 [59] in a global market. Quite a few native applications have turned into PWAs [14, 114, 119]. Figure 1 is a PWA example whose appearance looks after a native application, which can be simply installed through a browser like Figure 2.

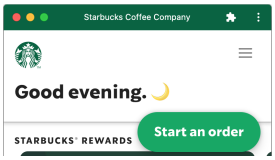

Figure 1: A real-world PWA example.

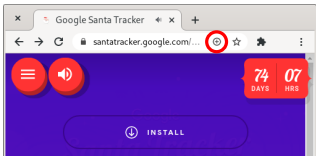

Figure 2: A PWA can be installed with a '⊕' button in a Chromium address bar (red-circled).

Unfortunately, a PWA has security problems that it inherits from a web application in addition to new problems due to sharing the same web runtime (e.g., a Chromium renderer that implements web APIs) across PWAs. A PWA is principally a web application with extended features. Thus, it could be vulnerable to conventional web attacks like cross-site scripting (XSS) [4, 23, 62, 81, 109] and supply chain attacks against dependent, external resources [106, 127]. Moreover, the impact of a successful exploitation can be exacerbated because PWAs installed on the same device hold identical attack surfaces. A compromised PWA can leverage *any web APIs* to attack the underlying web browser instance or other web applications. For example, several memory corruption vulnerabilities have been found in web APIs like `WebAudio`, `WebRTC`, and `WebUSB` [115–117]. A compromised PWA can freely access such vulnerable web APIs even though it does not require any of the web APIs at all for its intended operations (e.g., a Starbucks PWA does not require `WebUSB`). Our finding shows extensive API *bloating* in many PWAs; 90% of them utilize 15% or below of all web APIs available, which can pose a severe threat (§3.1).

One practical and effective mitigation technique to reduce such an attack surface is *software debloating* [8, 96, 97], which identifies and eliminates an unused functionality (i.e., web API). A browser-debloating technique is quite challenging due to the following reasons. First, the monolithic design of the browser makes it difficult to take a separate feature apart. Slimium [97] defines 164 distinct Chromium features containing at least 142,968 functions (around 40.1 MB), which is yet far from a complete feature set. Second, identifying the set of web APIs per PWA is non-trivial because ① static analysis of an untyped scripting language (i.e., HTML, CSS and JavaScript) is difficult [103]; ② a web browser may have its own (non-standard) web API; and ③ obfuscation techniques [12, 113] may contain redundant code for browser compatibility [82]. Further, our finding shows that the previous strategy [108] that eliminates high-cost and low-benefit web APIs cannot be directly applied to PWA because the web API distribution of a PWA differs from that of a conventional web application.

In this paper, we present DeView, a lightweight (but effective and efficient) debloating approach that reduces the attack surface of each PWA by narrowing down accessible web APIs. To this end, we devise two main techniques: *record-and-replay web API profiling* that identifies a set of web APIs a PWA demands and *compiler-assisted code debloating* that removes the web API entry functions. First, DeView records varying execution paths from unit tests and test cases written by the application developer, followed by collecting a set of web APIs by replaying the paths per PWA. Identifying code coverage from such test cases by the original developer assists in ensuring the intended features of an application are adequately covered (i.e., minimizing unexpected removal). Second, DeView instruments part of common libraries of a web browser at compilation time, producing a tailored version of those libraries that allows for the subset of web APIs. After removing 91.8% of unneeded web APIs with DeView, we can successfully thwart 76.3% of 478 known Common Vulnerabilities and Exposures (CVEs) pertaining to web API exploits, while features (from test cases) at a debloated PWA version have been seamless in our experiment.

The contributions of our work are as follows.

- We introduce DeView, a lightweight and practical PWA debloating approach that reduces its attack surface by trimming unnecessary web APIs.
- We design and implement the prototype of DeView with the two key techniques: record-and-replay web API profiling, irrespective of the complexity and implementation-specific features of a web browser, and compiler-assisted code debloating that eliminates the entry of unneeded web APIs.
- We evaluate DeView over 114 popular real-world PWAs. Our empirical results demonstrate the practicality and effectiveness of DeView by discarding over 90% of unneeded web APIs, thereby defeating 76% of known exploits.

We have open-sourced DeView[1] to foster the field of software debloating in the future.

2 BACKGROUND

This section describes a progressive web application, web API, and web interface description language.

Progressive Web Application. A PWA [92] is a new web technology to build a platform-independent universal application. It offers ① handy installability (i.e., one-click installation from a browser in Figure 2), ② smooth transition at a low cost from a web to a native application, ③ quick responsiveness, and ④ an indistinguishable appearance from a native application [101]. The key enablers of a PWA are the following HTML5 web technologies. First, `Service Worker` [7, 102] represents a worker thread that runs a particular JavaScript file in the background, allowing for caching PWA contents. A PWA leverages the worker as a network proxy to handle push notifications, updates, and network requests even under an inactive status. The events that a PWA can monitor via the worker include `installing`, `installed`, `activating`, `activated`, and `redundant`. Second, a `push notification` allows a remote server to directly push and notify a message to its recipients even when a PWA is offline. Third, `Web App Manifest` specifies a JSON file containing various PWA information (e.g., name, icon, launching URL), making a web application installable and discoverable. Lastly, client-side storage (i.e., `IndexedDB`, `Cache Storage`) allows a PWA to store data in persistent storage.

Web API and Security. A web API is a communication interface between a web application and a web browser. PWAs utilize the standard web APIs [118], encompassing both dynamic JavaScript APIs and other static APIs that handle HTML tags and CSS properties [31]. Oftentimes, adversaries attempt to exploit vulnerable web APIs via varying web attacks such as XSS [75] and Universal

[1] https://github.com/shivamidow/deview

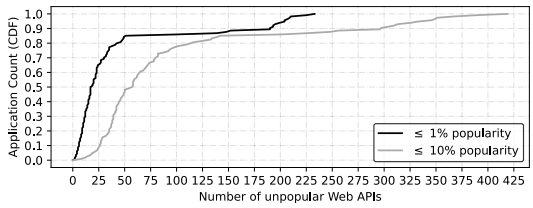

Figure 3: Cumulative distribution of PWAs according to the popularity of required web APIs. Adopting unpopular web APIs in a PWA is common.

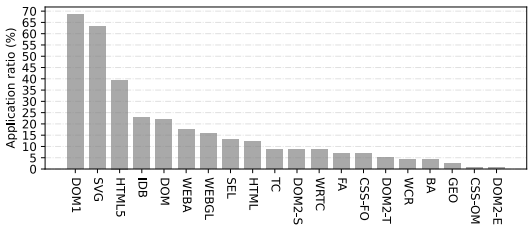

Figure 4: Distribution of PWAs according to the classes of required web APIs suggested by Snyder et al. [108].

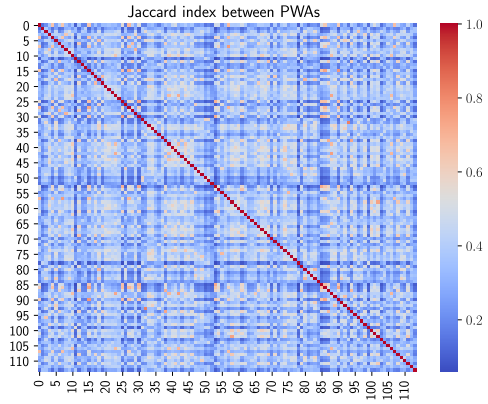

Figure 5: Jaccard indexes of PWA pairs. The blue color dominates the heatmap, meaning most PWA pairs in our dataset do not have similar web APIs in common.

XSS [81], which mostly leverage JavaScript. Moreover, HTML tags and CSS properties can be weaponized [4, 23] as well. A successful web attack can disarm the browser's security protection mechanisms like the same-origin policy (SOP) and underlying sandbox, allowing adversaries to exfiltrate the victim's sensitive data or lure a victim into visiting a suspicious page [56]. In this respect, we aim to restrict web APIs that each PWA can use to reduce the potential risk of a compromised PWA.

Web Interface Description Language. `WebIDL` [74] is an interface description language that specifies each web API. While W3C standardizes specifications on `WebIDL`, each web browser vendor extends it with customized attributes and features [25, 28, 30]. `WebIDL` plays a pivotal role in binding the entry point of a web API (i.e., HTML tags, CSS properties, JavaScript APIs) to an underlying native implementation (mostly written in C++). In this paper, we leverage `WebIDL` to find the corresponding implementations of a web API for further debloating.

3 PRELIMINARIES

3.1 Web APIs in a PWA

We investigate the usage of web APIs (i.e., distribution and similarity) in 114 PWAs (§6.1), comparing it with that of a conventional web application. This section describes the direction that motivates us to propose a debloating technique tailored for a PWA.

Popularity Distribution of Web APIs used by PWAs. Previous studies [97, 107, 108] find that most legitimate web applications use similar web APIs. Thus, they can secure web browsers by removing unpopular web APIs without significant incompatibility problems. However, such a popularity-based strategy does not work for PWAs because they frequently use unique web APIs. We investigate the distribution of web APIs that 114 different PWAs adopt. We collect 8,249 web APIs in total from the popularity data of features (`featurepopularity.json`) and CSS (`csspopularity.json`) from `chromestatus`[2] [22] and check how or whether the collected PWAs use them. Figure 3 shows the cumulative distribution of the PWAs according to the adoption of unpopular web APIs. We define a non-popular web API when it has been adopted by less than or equal to k% (e.g., k=1 and k=10) of the whole PWA applications in our dataset. We confirm that around 50% of the PWAs employ at least 20 web APIs that hold ≤1% in popularity and at least 60 web APIs that hold ≤10% in popularity.

Adoption of a Previous Approach. According to Snyder et al. [108], high-cost and low-benefit web APIs are removable because web applications barely use certain classes of high-cost web APIs (e.g., Scalable Vector Graphics (`SVG`), Web Audio (`WEBA`), Web GL (`WEBGL`), Web RTC (`WRTC`)) in most cases. We analyze 114 PWAs to know whether Snyder et al.'s finding is applicable to them. To this end, we classify the unpopular web APIs (≤1% in popularity) into the same categories as Snyder et al. However, our observation indicates that their finding does not work for the 114 PWAs because, unlike typical web applications, they frequently adopt high-cost and low-benefit web APIs (Figure 4). For example, 63.16%, 17.54%, 15.79%, and 8.77% of the PWAs use `SVG`, `WEBA`, `WEBGL`, and `WRTC`, respectively.

Unique Web API Usage Between PWAs. We study whether the web API usage patterns of 114 PWAs are similar to each other. We use the Jaccard index [121] to measure their similarity. The heatmap in Figure 5 illustrates Jaccard indexes between each unique pair of the PWAs. A red cell (Jaccard index is 1) represents that a pair of PWAs employs an identical set of web APIs, whereas a blue cell (Jaccard index is 0) represents that a pair of PWAs employs a completely different set of web APIs. The Jaccard index is 0.36 on average, while the minimum and maximum values hold 0.06 and

[2] `Chromestatus` officially collects the statistics of web API usages from anonymous Google Chrome users during the last 24 hours. We extract `property_name` and `day_percentage` fields of each web API from the `json` file. As the `property_name` represents a Chromium's internal function, replacing it with a standard web API.

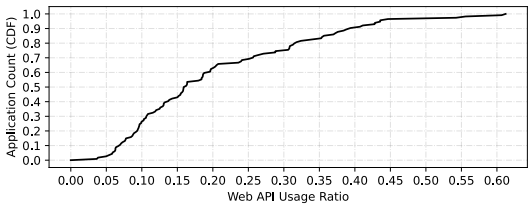

Figure 6: Cumulative distribution of PWA web API usage ratios over total web APIs used by at least one PWA.

0.82, respectively. The result shows that most PWAs have *a unique set of required web APIs*.

Building Common Debloated Browsers for PWAs. Like previous studies [97, 107, 108], one might want to build a single or a small number of web browser instance(s) that can support all or most PWAs but are less bloated than a pristine web browser. Figure 6 depicts the cumulative distribution of the web API ratio where each PWA uses over the union of web APIs that at least one PWA uses. The 114 PWAs in our dataset adopt 3,296 distinct web APIs in total, but approximately 90% of the PWAs utilize less than 40% of the 3,296 web APIs. This implies that a single debloated browser that covers all PWAs (i.e., supports 3,296 web APIs) is still highly bloated (i.e., at most 40% of supported web APIs are used in general).

Summary of Our Findings. The above findings conclude that ① the web API usage of PWAs considerably differs from each other as well as that of a web application, ② a debloating strategy based on web API popularity does not work for PWAs, ③ unpopular web APIs of PWAs play a more pivotal role than those of typical web applications, and ④ each PWA necessitates a different debloating rule due to its unique web API usage.

3.2 Challenges

Monolithic Design. A browser keeps including additional features in the form of web APIs [22, 32] to meet users' demands. Modern web browser vendors create new web standards beyond just complying with existing ones [123] while being compatible with legacy and non-standard features, which inevitably enlarges an attack surface. Modularizing each web API would be ideal for security by reducing the impact of a single web API compromise on others, but a vast number of web APIs make such modularization impractical for performance concerns. A recent effort for attack surface reduction is `Permissions Policy` [26] by letting a web application selectively enable needed web APIs on demand. However, the policy feature is insufficient because ① not all web browsers support it [40], ② it cannot allow one to configure a fine-grained feature compartment, and ③ a web API exploit can potentially bypass it.

Precise Identification of Web APIs. For successful confinement, obtaining the precise list of web APIs that each PWA requires is necessary. However, in general, even developers cannot statically figure out comprehensive web API usages due to the dynamic nature of writing web content and browsers. First, a static analysis of JavaScript is challenging because it allows one to ① override a function and object, ② execute a string with an `eval` function at runtime, ③ manipulate both DOM and CSS on the fly, and ④ obfuscate codes with a mangler or compressor [12, 113]. Although the static analysis of a scripting language is not impossible, it is quite slow [63]. Second, the browser-specific behavior of a certain web API can trigger a different set of web APIs. For example, `polyfill` [82] enables one to address issues with branch statements that handle vendor prefixes [85] by having different browsers implement an identical functionality in a different manner (i.e., different web API invocation). A recent study from Sarker et al. [103] shows that static analysis cannot precisely analyze mangled web APIs. Therefore, we adopt a dynamic approach that achieves both high accuracy and scalability in practice.

3.3 Threat Model

The following shows a conceivable attack scenario. Assume that a victim uses an Uber PWA to commute. Before choosing a driver, one checks a driver's reputation for safety reasons. Suppose that a vulnerability were present in a text input form (due to the lack of input sanitization) for commenting on a driver at the Uber PWA. In that case, an adversary can exfiltrate sensitive data by injecting a malicious code (e.g., `WebUSB` API). For example, a secret token stored in the victim's USB drive could be leaked when the victim reads the adversary's comment. In this scenario, an adversary conducts a *web attack* against a PWA or the underlying browser instance with the following two steps. First, the adversary prepares a malicious code in the context of a victim PWA (e.g., via code injection or reuse) by exploiting a vulnerability in the PWA, a browser engine, or a server-side code. Next, the adversary executes the malicious code that misuses a web API, being able to take complete control over the PWA or the browser engine.

This paper focuses on preventing or mitigating this critical second step by restricting each PWA's web API. The malicious code might exploit other vulnerabilities unrelated to web APIs, but it is beyond this paper's scope. Moreover, DeView aims to prevent web API misuse attacks from happening inside the PWA scope [87]. So, any attack with external resources that are loaded in the main frame or the iframe within a PWA can be prevented if its exploitation entails a disallowed web API. However, such an attack out of the PWA (e.g., a separate window) is out of our threat model. Like other debloating or sandbox mechanisms, PWA developers determine which web APIs each PWA shall use.

4 DEVIEW DESIGN

In this section, we describe the design of DeView.

4.1 Design Overview

DeView largely consists of three components: ① web browser instrumentation that identifies entry web API functions (by a browser vendor), ② web API usage profiling based on a record-and-replay technique (by a PWA developer), and ③ on-demand browser engine debloating (by an end-user program).

Figure 7 shows the overall processes of DeView. ❶ A PWA developer records user interactions (using pre-defined unit test cases) with a web browser behavior recording tool, which translates the behaviors into web browser instructions. The developer replays the

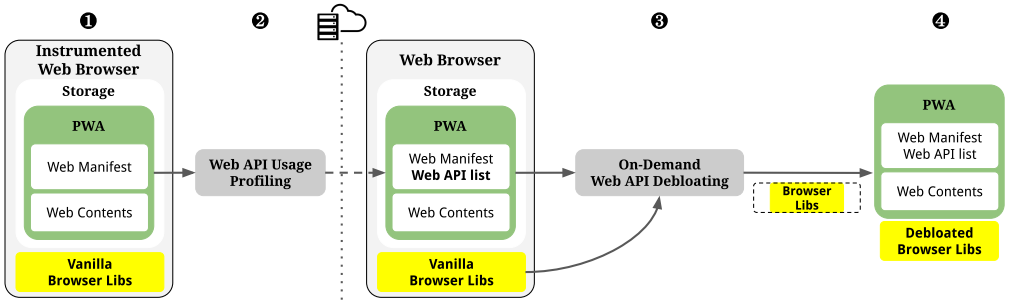

Figure 7: Overall procedure of DeView. ❶ DeView instruments a web browser to profile exercised web APIs. ❷ A PWA developer dynamically collects a list of needed web APIs. ❸ An installer program at the end user debloats the browser library along with the list of required web APIs. ❹ DeView launches the debloated PWA.

recorded instructions on an instrumented browser instance to monitor web API usages and generates a list of required web APIs for the PWA. ❷ A server sends a PWA along with its web API list to an end-user upon an install or update request. ❸ An installer program at the end-user generates the debloated version of a browser engine library with the web API list for the PWA. ❹ Running a PWA simply launches a browser with the debloated library. DeView re-performs the above debloating processes in case of a PWA update via the `Service Worker`'s install event [7].

4.2 Web API Library Separation

DeView restricts each PWA's web API by debloating the underlying web browser. However, debloating the whole web browser is challenging because it is huge, complex software [97]. Instead, DeView *decouples* a core browser engine that contains the entry points of all web APIs from the main browser and debloats the core browser engine as shared libraries (i.e., trim the entry points of unnecessary web APIs §4.5). On the end-user side, DeView maintains debloated libraries per PWA to confine each PWA's web API usage separately and to refine or replace the libraries if a PWA or its web API list is changed.

4.3 Web Browser Instrumentation

We instrument a web browser to identify exercised web APIs with PWA profiling. The entry function of each web API resides in a part of the browser engine libraries, which will be debloated on demand. DeView first recognizes native functions that correspond to an individual web API based on a `WebIDL` binding rule, followed by instrumenting the web browser for web API profiling. In particular, DeView maintains the location of native functions and their sizes for the instrumented browser, which are utilized for later debloating. Note that the instrumentation is a one-time process for a target browser version.

4.4 Web API Usage Profiling

DeView profiles what web APIs are required to run a PWA when it is newly created or updated. DeView first runs a PWA with given unit test cases or a manual test scenario while recording all browser interactions using a `Headless recorder` [17]. Then, it replays the recorded browser interactions on the instrumented web

```
1  const puppeteer = require('puppeteer');
2  (async () => {
3    const browser = await puppeteer.launch();
4    const page = await browser.newPage();
5
6    // Launch the starbucks app in the viewport of 952x1021
7    await page.goto(
8      'https://app.starbucks.com/?utm_source=homescreen');
9    await page.setViewport({width: 952, height: 1021});
10
11   // Wait for a text link and click it.
12   await page.waitForSelector('.flex > .globalNav > .textLink');
13   await page.click('.flex > .globalNav > .textLink');
14
15   // Wait for a form, activate it by click,
16   // and type 30332 in the form.
17   await page.waitForSelector('.header > .controls > .form');
18   await page.click('.header > .controls > .form');
19   await page.type('.header > .controls > .form', '30332');
20
21   await browser.close();
22 })();
```

Figure 8: A simplified example of recorded browser instructions for the `puppeteer`. DeView sequentially replays the instructions for reproducing keyboard and mouse events (Line 18–19), enabling one to profile web APIs after a sign-in.

browser using a browser automation tool, `Puppeteer` [38]. When a web API is firstly accessed, the instrumented web browser updates a bit array where each bit represents whether a corresponding web API is accessed. After the replay finishes, DeView creates a list of required web APIs based on the final bit array.

Our record-and-replay profiling technique has advantages over conventional dynamic analysis based on event-driven unit tests and static analysis. First, it is scalable to various platforms because recorded browser interactions can be replayed on various web browsers that conform to the `WebDriver` specification [110] while even capturing vendor- or version-specific non-standard web API usages. Second, it is reliable and robust for profiling web API that a PWA exercises. As recorded behaviors can be consistently executed with `Promise`, the profiling result does not suffer from any race condition issue between a layout and an input, or the effect of non-deterministic behaviors within a browser engine [97]. Third, it is handy to carry out a comprehensive test for identifying exercised web APIs even behind a complex user interface, authentication, or payment process by simply recording user interactions (e.g., keyboard typing, mouse clicking).

Example. Figure 8 shows a simplified code snippet for profiling the `Starbucks` PWA through `Puppeteer` [38]. It creates a web browser instance, then loads the initial page of the `Starbucks` PWA (Line 9). Line 12 and 13 indicate that it clicks on a text link with a selector whose ancestors are `flex` and `globalNav` in the DOM (i.e., `.flex > .globalNav > .textLink`). Line 17–19 represent a behavior of typing a number, 30332, after clicking a form element with `.header > .controls > .form` as its selector. Line 21 closes the browser instance. Note that the `await` operator from Promise [84] ensures that all instructions run synchronously, explicitly waiting for a DOM node corresponding to a given selector before further actions (e.g., clicking or typing). In particular, the `waitForSelector` method aids in avoiding a race condition problem between input event processing and DOM operation, which event-driven unit-test-based profiling suffers from otherwise. With the Starbucks PWA, DeView can catch 591 web APIs used for rendering in a PWA startup (53% of the full web APIs we get from thorough manual tests) and examine multiple aspects of the application at once with a single click or test input. Note that DeView can obtain exercised web APIs with pre-recorded behaviors flawlessly with a minimum effort—that is, DeView solely consumes the amount of CPU and memory resources as much as having a single web browser instance.

4.5 On-Demand Debloating

DeView debloats browser engine libraries for each PWA when it is newly installed or updated with a revised list of required web APIs. This on-demand debloating relies on ① the boundary information (i.e., location and size) of each web API's entry function in the libraries received from the browser vendor (§4.3), ② a list of required web APIs received from the PWA developer (§4.4), and ③ modified `Service Worker` to intercept installation events [7]. DeView removes the entry functions of unnecessary web APIs from the browser engine libraries by replacing the corresponding low-level machine code with a software interrupt instruction [120]. Instead of this strict protection with an immediate crash, DeView can adopt a gentle fallback approach (§7). Finally, DeView maintains slimmed browser engine libraries for each PWA and loads them when it launches the PWA (e.g., via `LD_PRELOAD`).

5 IMPLEMENTATION

We implement DeView on top of the Chromium version of 80.0.3987.0 (r722234) built with LLVM/Clang 10.0.0 (pre-release version), running on Fedora 32 (kernel version 5.8.10). Table 4 shows Chromium compilation options that affect the number of web APIs. Note that `proprietary_codecs` and `ffmpeg_branding` are enabled by default for playing a protected media (e.g., DRM). For record-and-replay web API profiling, we used `Puppeteer v2.1.1` and `Headless Recorder v0.8.0`.

JavaScript Execution with Puppeteer. We employ `Puppeteer` for automatic browser control and sequential JavaScript execution, simplifying our record-and-replay web API profiling for DeView. We added 705 lines of Python, 1,066 lines of C++, and 142 lines of JavaScript code to `Puppeteer`.

WebIDL and Web API Entry Functions. A majority of modern web browsers have a bridge or binding layer between front script engines (i.e., HTML, CSS, and JavaScript) and underlying native (C++) implementations to enable web APIs to interact with corresponding native functions. Browser vendors use `WebIDL` to specify such interfaces, write them in `WebIDL` files and run a `WebIDL` parser against the files to automatically generate bridge functions that follow consistent naming rules. We investigate the naming rules of Chromium's `WebIDL` parser (i.e., Blink IDL) to obtain a comprehensive list of all web API entry functions. Table 1 shows the rules that we employ to seek the entry point of a target web API native function. For example, we can enumerate entry points of all HTML web APIs by searching native functions whose names end with `Constructor` in `html_element_factory.cc`.

LLVM Passes. We develop two LLVM passes for web API marking and dynamic profiling. The web API marking pass identifies the boundary information (location and size) of all web API entry functions. The dynamic profiling pass identifies requested web APIs during a PWA's execution. This pass maintains a bit array in a shared memory where each bit represents whether a corresponding web API has been requested.

Browser Library Debloating. For each PWA, the debloating process generates debloated versions of Chromium's two libraries containing all entry functions of native web API implementations: `libblink_core.so` and `libblink_modules.so`. The process relies on the boundary information and a set of required web APIs collected via dynamic profiling, which PWA developers provide. We modified `Service Worker` to debloat the libraries only when a PWA is installed or updated. (i.e., the `OnStoreRegistrationComplete` method in the `ServiceWorkerRegisterJob` class selectively forks the debloating process.) As a final step, our specialized Chromium loads debloated libraries when launched via the `LD_PRELOAD` environment variable.

6 EVALUATION

We set up the following four research questions to evaluate DeView.

- **RQ1.** How many web APIs can DeView remove in a debloating browser engine (§6.2)?
- **RQ2.** How effectively does DeView prevent possible attacks (§6.3) with a case study (§A.2)?
- **RQ3.** How much code coverage can DeView achieve in finding exercised web APIs (§6.4)?
- **RQ4.** What are the performance overheads of DeView (§6.5)?

Experimental Environment. We evaluate DeView on the Linux machine (kernel v5.8.13) equipped with Intel Core i7-8565U CPU (4 cores, 1.80 GHz) and 16 GB of memory.

6.1 Dataset

PWAs. We gather real-world PWAs from the Alexa Top 100 US sites [5] and other online resources [6, 52, 95] due to the absence of a central repository for PWAs. We installed them with our modified Chromium that can log web APIs usage (§4.4). Note that we excluded non-installable or malfunctioning PWAs, which mostly arise from proprietary software for copyright protection or codec [15]. In total, we have 114 PWAs. To identify necessary web APIs for each PWA, we navigate all available user interfaces of PWAs (e.g., click buttons, fill out input fields, and scroll up and down pages)

Web API Type	Target Files	Class Naming Rules	Target Function Naming Rules

HTML	`html_element_factory.cc`
	`mathml_element_factory.cc`
	`svg_element_factory.cc`
			`[.*]Constructor`

CSS	`shorthands.cc`
	`shorthands_custom.cc`
	`longhands.cc`
	`longhands_custom.cc`
			`[.*][ApplyInherit | ApplyInitial | ApplyValue | CSSValueFromComputedStyleInternal | GetJSPropertyName | InitialValue | ColorIncludingFallback | ParseSingleValue | ParseShorthand | ConsumeAnimationValue | ConsumeFont | ConsumeImplicitAutoFlow | ConsumeSystemFont | ConsumeTransitionValue]`

JavaScript	`v8_[.*]`	`![.*]v8_internal`	`[.*][MethodCallback | AttributeGetterCallback | AttributeSetterCallback | ConstructorCallback]`

Table 1: LLVM pass rules to identify web API entry functions based on their names.

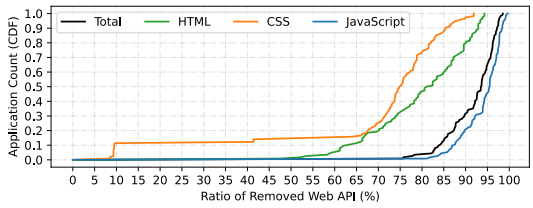

Figure 9: CDF of removable web APIs ratio in our dataset. On average, 91.8% of web APIs are removable.

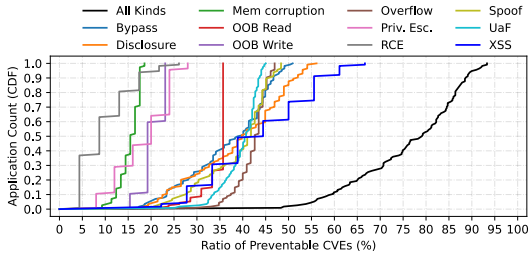

Figure 10: CDF of the ratio for CVEs preventable by DeView. It can prevent 76.3% of 478 CVEs on average (Table 5).

to trigger as many web APIs as possible. This approach is similar to Snyder et al.'s feature detection [108]. In addition to it, we use `gremlins.js` [73] to automatically detect any web APIs we might miss. Detailed statistics can be found in §3.1.

CVEs. We collect real-world web exploits which rely on web APIs so that DeView might be able to mitigate them. First, we collect 1,035 CVEs from Chrome release notes [21] over the last five years. Second, we visit each CVE's bug ticket in `crbug.com`, the official bug tracking system for Chromium, to find CVEs associated with any web APIs that our PWAs use. Among them, we find 478 CVEs relevant to web APIs. Some of them are initiated with web APIs, and some others trigger final actions with web APIs. Note that we exclude CVEs whose information has yet been revealed. Finally, we classify them into 11 types based on bug description and proof-of-concept: bypass, information disclosure (Disclosure), memory corruption, out-of-bound read (OOB Read), out-of-bound write (OOB Write), overflow, privilege escalation (Priv. Esc.), remote code execution (RCE), spoofing, use after free (UaF), and cross-site scripting (XSS). Interested readers refer to Table 6 in Appendix (§A.4). Note that we discard CVEs that are irrelevant to web API exploits as described in our threat model (§3.3). For instance, `CVE-2019-5777` is a URL spoofing attack that attempts to mislead users to `www.o2.co.uk` with a Unicode `U+0B20` that confuses one with the letter 'O'. Such a spoofing attack does not entail any web API, which is out of DeView's scope.

6.2 Removable Web APIs

We assess DeView's effectiveness in removing web APIs with the 114 PWAs. We count the number of web APIs that can be eliminated from the browser engine for each PWA by subtracting an exercised web API list per PWA from the entire web API list in our Chromium.

Results. Figure 9 illustrates the result where the x-axis represents the ratio of removed web APIs, and the y-axis represents the cumulative fraction of the 114 PWAs. DeView eliminates 91.84% of web APIs (i.e., a combination of HTML, CSS, and JavaScript APIs) on average, ranging from 75.54% to 98.53%. As a breakdown result, DeView successfully shrinks 79.95% of HTML (47.39%–94.31%), 68.41% of CSS (8.47%–91.86%), and 94.03% of JavaScript APIs (81.12%–91.62%) on average, respectively. We display the results for the whole PWAs in Table 5.

6.3 Security Benefits

We evaluate the security benefits of DeView by estimating how many CVEs can be mitigated by removing unnecessary web APIs. Figure 10 shows the ratio of preventable CVEs for 114 PWAs. DeView blocked 76.33% of web API-relevant exploits on average, ranging from 48.33% to 93.31% (See Table 5 in §A.3). While most CVEs are `JavaScript` exploits (i.e., 420 of relevant CVEs), we also observe non-JavaScript attacks that exploit the static web APIs (`HTML`: 48 CVEs, `CSS`: 10 CVEs). The effectiveness of DeView slightly differs across attack types: DeView is effective against bypass and XSS, whereas less effective in defeating RCE or other memory-related attacks. That is because RCE and memory attacks mostly exploit non-web API methods, such as JavaScript language natures (e.g., `TypedArray`, `RegExp`, `wasm`) or browser infrastructure (e.g., extensions, PDF, protocol handler).

6.4 Code Coverage

In this section, we discuss the effectiveness of DeView's Web API profiling in terms of code coverage. We choose three popular PWAs: Starbucks, Telegram [112], and XSound [57] from eCommerce, social media, and productivity categories, respectively. With the DevTools coverage [11], we compare the code coverage of DeView

	Code coverage						#web APIs		
	JavaScript			CSS					
PWAs	**Used/Found (MB)**	**%**	**#Files**	**Used/Found (kB)**	**%**	**#Files**	**JavaScript**	**CSS**	**HTML**
DeView									
Starbucks	2.60/4.19	62.18	59	36.7/166.4	22.08	9	764.33	154	57
Telegram	1.02/2.63	38.67	1	66.3/237.3	27.96	2	383	149	43
Xsound	0.28/0.51	55.42	5	24.0/27.5	87.06	3	502.33	145.33	47
gremlins.js									
Starbucks	1.52/3.14	48.63	33	17.4/143.1	12.17	6.33	660	148	43.33
Telegram	0.99/2.63	37.73	1	55.6/237.3	23.45	2	382.33	143	40
Xsound	0.28/0.51	55.01	5	21.1/27.5	76.52	3	456.67	145.67	46
DeView + gremlins.js									
Starbucks	2.60/4.19	62.32	59	36.9/165.1	22.38	9	768	156	58
Telegram	1.03/2.63	39.25	1	74.3/237.3	31.30	2	441	149	51
Xsound	0.29/0.51	56.62	5	24.0/27.5	87.06	3	515	147	47

Table 2: Comparisons of DeView and `gremlins.js` on code coverage and the number of discovered web APIs for three popular PWAs. Each experiment was conducted for four minutes and repeated three times. DeView surpasses `gremlins.js` in both code coverage and web API discovery. Combining DeView and `gremlins.js` promotes both code coverage and web API profiling.

for the three PWAs with that of `gremlins.js` [73], the well-known monkey test framework for a web application. We recorded each PWA with ordinary activities and replayed the recorded ones (i.e., encoded instructions) with our instrumented Chromium (§4.3). For monkey testing, we injected the local `gremlins.js` script into each PWA to circumvent their CSPs, followed by starting a monkey test in the same instrumented Chromium with `Puppeteer` APIs (e.g., `page.addScriptTag` and `page.evaluate`). We monitor each PWA once its main page has been loaded. In the case of Telegram, we had to sign in to collect sufficient code for testing. Note that each experiment was performed for 240 seconds and repeated three times. It is worth noting that achieving full code coverage on PWAs is often infeasible in profiling web APIs because ① a certain code block may be redundant for compatibility across a different browsing environment (i.e., `Polyfill`), ② an exception handling would be unreachable until encountering an error, and ③ a third-party library (e.g., `jQuery`) can be inevitably imported.

Results. Table 2 shows that DeView outperforms the `gremlins.js`-based monkey test in both code coverage and web API profiling, and the code coverage and the web API profiling efficacy could be maximized when we combine DeView and the monkey test. For example, in the case of Starbucks PWA, DeView examined 62.18% of JavaScript code and 22.08% of CSS code from 59 JavaScript and 9 CSS files. In contrast, `gremlins.js` only covered 48.63% of JavaScript code and 12.17% of CSS code from 33 JavaScript and 6 CSS files. Telegram's code coverage was lower than those of the two other PWAs because it consisted of a single, sophisticated JavaScript file supporting various browsers and platforms. Many of its JavaScript functions were selectively executed according to the underlying environments. The code coverage of CSS was lower than that of JavaScript in general because similar to the case of Telegram, PWAs usually had bloated CSS files supporting various browsers and platforms. Furthermore, DeView found more JavaScript, CSS, and HTML web APIs than those found by `gremlins.js` (15.8%, 4.1%, and 31.5% more, respectively, in the case of Starbucks). DeView was better than the monkey testing because its record-and-replay-based profiling is guided. Since a developer can accurately program the features to be tested on specific pages with correct input values, DeView can maximize the test coverage without suffering from pitfalls that the monkey test encounters. For example, we observed that the monkey test frequently got stuck on a certain page requiring valid input for proceeding (e.g., login forms). Moreover, it frequently revisited a page profiled before and often deviated from the scope of the profiling target by accidentally clicking out-links and never coming back on track. Despite these shortcomings, we can consider the monkey test as a supplementary method to capture web APIs that DeView might miss. As shown in Table 2, when we applied `gremlins.js` after DeView, the code coverage and the number of found web APIs improved for Telegram and XSound, which are single-page applications. Thus, we conclude that using both approaches together effectively finds required web APIs.

6.5 Performance Overheads

	Starbucks	Telegram	XSound
CPU (%)	29.02	13.54	27.59
Memory (MB)	390.49	245.68	465.78

Table 3: Performance overheads for profiling web APIs with three PWAs.

Developer-Side Overheads. For web API profiling, DeView records a PWA's execution with `Headless recorder` [17] and replays the recorded instructions with `Puppeteer` [38] on top of our instrumented browser. We confirmed that both `Headless recorder` and `Puppeteer` incurred negligible CPU and memory overheads. Table 3 shows the CPU and memory requirements (in the experimental environment) for running three different PWAs, which may vary depending on the characteristics of PWAs. Note that the memory overheads come from the sum of private memory footprints [18] by leveraging the Chromium's task manager and `chrome://tracing` [24], considering all sub-processes (e.g., renderer, browser, GPU, network, and audio).

User-Side Overheads. The overheads from an end-user arise from debloating browser engine libraries (i.e., copying the vanilla

libraries and removing unnecessary web API entry functions). The experiment of 10-times debloating for each PWA took 0.24 seconds on average (max = 0.29 seconds). The size of the whole debloated libraries pertaining to web APIs per PWA is approximately 68 MB.

7 DISCUSSION

In this section, we discuss a few limitations of DeView along with future work.

Generality. DeView debloats web APIs by leveraging compiler-assisted annotations on each entry function of a web API that has been collected from replaying user behaviors. Hence, in general, DeView 's approach would be applicable to other modern browsers (i.e., `WebKit`, `Gecko`, and `Chromium` variants) because they also follow the `WebIDL` protocol [74] in binding web APIs to low-level implementations (§2) and support the `WebDriver` interface [110] for the platform- and language-neutral automated testing.

Representativeness of PWAs. Due to the absence of a central repository for PWAs, we collect varying PWAs (from different categories, including travel and game) of our choice. As our dataset does not fully represent all PWAs, removable web APIs and preventable CVEs may vary accordingly. For better representativeness of PWAs, it is possible to crawl them from the Alexa Top websites by listening to a `beforeinstallprompt` event [39, 86] that informs the availability of a PWA. However, the `beforeinstallprompt` event is an experimental feature, which leaves it as future work.

Breakage Cost. DeView disables web APIs by overwriting their entry functions with software interrupt instructions (i.e., `INT` in the `x86` architecture) to suspend potential exploitation immediately. This strict policy potentially results in a bad user experience if unexpected crashes occur due to updated third-party libraries or dynamically-loaded advertisements. To alleviate such unwanted experiences, DeView provides a configurable fallback mechanism. For example, instead of using software interrupt instructions at debloated web API call sites, DeView can return a legitimate error code to a caller, display a warning message in detail, or simply ignore an exception depending on the policy from which a user chooses. In addition, if a PWA needs to access a debloated web API for a critical but unanticipated operation (e.g., browser extensions), a user still has a choice from either loading the PWA with vanilla libraries for compatibility or disabling a problematic function for security. This is similar to Firefox's Troubleshoot Mode [29], which reloads a troublesome webpage in a pristine state without extensions, caches, and cookies.

Non-Web API Attacks. DeView does not cover web exploits against built-in JavaScript objects, properties, and methods [83] that do not rely on web APIs (e.g., `Date`, `Math`, `RegExp`). A potential countermeasure for them would be to eliminate the corresponding implementation from the JavaScript engine. However, it is non-trivial because removing a primitive type of JavaScript language affects other internal implementations. We leave this to future work.

Needed Web API Attacks. DeView cannot cover the case where an exploit compromises a PWA only with required web APIs. Nevertheless, as shown in §3.1, 90% of PWAs use at most 15% of web APIs. Thus, the remaining web APIs that a compromised PWA can use are fairly limited.

Limited Code Coverage. Exercising every code path of a web application is very challenging due to third-party libraries (e.g., `jQuery`), exception handling routines, and compatibility (e.g., `polyfill`) issues, and so is DeView. Since web API profiling relies on dynamic analysis, it may suffer from limited code coverage due to an incomplete set of test cases, potentially missing some required web APIs. Producing a complete and sound test case is yet another research topic [103], which is beyond this paper's scope. Still, our experiment shows that a semi-automated approach has been sufficient to instrument a debloating version that works seamlessly. Although a fuzzing technique helps to increase overall coverage, it is susceptible to exploring a context-sensitive feature like a sign-in. Utilizing both static and dynamic analysis for better coverage is part of our future research. Also, to overcome this limitation, DeView can leverage client-side error logs to further collect unidentified web APIs, followed by updating debloated browser libraries accordingly.

Storage Overheads. As DeView generates a debloated library version per PWA, it consumes additional storage proportionally to the number of installations. However, this storage overhead is still smaller than those of Electron apps (~120 MB) that must contain the entire copy of Chromium [64, 90]. DeView can reduce this storage requirement by adopting memory de-duplication or masking out unneeded web APIs in a library at load time, which remains part of our future research.

8 RELATED WORK

Reducing attack surfaces and diversifying shared libraries are classic problems in system security that have been the subject of extensive previous work. We will discuss them in groups of different targets.

Operating System Debloating. An operating system contains a large spectrum of modules providing functionalities throughout the system stack, including the kernel, system calls, system libraries, etc. Gu et al. [48] propose FACE-CHANGE dynamically customizing kernel code at the basic block level when the running application is changed; thus, each application accesses the required minimized kernel code at runtime. Similarly, Zhi et al. [128] leverage hardware-assisted virtualization to enable or disable the executable permission for required or unneeded code pages for different running applications. Both Kurmus et al. [68] and Kuo et al. [67] profile a workload to obtain used kernel configurations and statically tailor the kernel by only enabling the required configurations at compile time. In addition to directly shrinking kernel code size, DeMarinis et al. [37] and Ghavamnia et al. [44, 45] limit an application's access to system calls. Some systems have also been designed to create minimal containers to provide limited operating system resources (e.g., CPU, memory) to target applications [50, 54, 100]. Compared with these systems, DeView reduces web APIs as attack surfaces from web browsers on which PWAs are frequently updated.

Library Debloating. Although a library wraps up a bundle of functions providing fruitful features, most applications just use a subset of them. This results in the library being bloated. CodeFreeze [88] conservatively analyzes a binary and eliminates functions imported but never used by the binary from libraries at load time. PieceWise [99] recompiles a library to figure out call dependencies and

function boundaries and save them into the final binary as supplementary information for future debloating. It also modifies the loader to invalidate library code unnecessary for target applications. Nibber [2] disassembles libraries that an application requires, constructs function call graphs in order to identify unreachable code for the application, and then nullifies dead code by directly rewriting the libraries. Even though DeView also debloats shared libraries of a web browser, it is more flexible and scalable than the aforementioned work. This is because DeView regenerates a newly debloated copy of browser libraries whenever the target PWA is either updated or installed, rather than freezing the debloated libraries forever after doing static analysis once.

Web Browser Debloating. A web browser consists of large and complex source code to provide users a complete set of tools necessary to explore the world wide web and satisfy web developers' various demands with rich web features. Previous work related to web browser debloating systems is the closest to our work. Snyder et al. [108] evaluate the costs and benefits of allowing a website to use each web feature, then restrict the website from accessing risky features by utilizing JavaScript `Proxy` objects in their browser extension. However, this approach is not only vulnerable to static web API (i.e., HTML tags and CSS properties) based attacks [4, 23, 111] but also easy to bypass because Web APIs' entry points and actual implementation still reside in computer memory [60]. Moreover, this approach cannot be applied to PWAs since even unpopular Web APIs can play a pivotal role for a specific PWA, as shown in Figure 4. Slimium [97] is the first work that succeeds in debloating browser-scale complex software. It removes unnecessary functions for a landing page at the feature granularity from a Chromium binary based on a predefined feature-code map (with a semi-automatic analysis), instrumenting a slim version of Chromium per website. DeView has different motivations and objectives. It pursues a general way to confine accessible web APIs per PWA and seeks incorporated Web APIs in a PWA with user interactions. In addition, a study by Lee et al. [69] shows that unique features of PWAs, such as `Push Notification`, `Cache`, `Service Worker` can be weaponized by attackers. This indicates that PWAs can be exposed to additional security threats that traditional websites and web applications never suffer. DeView is the first web browser debloating system with a focus on PWAs.

Application Debloating. Researchers have proposed numerous systems for debloating applications. DamGate [19] proposes a framework using both static and dynamic analysis to build a call graph based on the allowed features at runtime it customizes. Meanwhile, Shredder [77] performs argument-level debloating against well-known APIs with the initial analysis of benign API parameters, followed by establishing a policy to narrow the scope of parameters. In a similar vein, Saffire [78] expands Shredder to create a hardened replica of a function with a restricted call invocation. Razor [96] comes up with four heuristic techniques to deduce more code paths that have not been exercised with given test cases but possibly needed for the desired functionality. Koo et al. [66] debloat partial code of an application relying on specific runtime configurations. Babak et al. [8] propose an approach using two levels: file-level and function-level to debloat PHP applications by identifying required code during interactions between a client and a server. Machine learning (e.g., deep learning and reinforcement learning) based debloating techniques also have been introduced. Hecate [125] makes use of both Recursive Neural Network (`RNN`) to obtain code embedding and Convolutional Neural Network (`CNN`) to identify feature-constituent functions for further debloating. Binary control flow trimming [43] proposes a contextual control flow graph (`CCFG`) that filters out unneeded features with a combination of runtime tracing, reference monitoring, and machine learning. CHISEL [51] leverages reinforcement learning to generate a debloating variant after unneeded feature elimination.

Software Diversification and Obfuscation. DeView employs a software hardening technique to raise the bar for attackers to seek and exploit security vulnerabilities in PWAs. Previous researchers tried to impede malicious reverse engineering attempts with various software obfuscation techniques at the code level [9, 10, 13, 20, 34, 55, 76, 79, 98, 105, 124] and hamper hostile problem analysis that purposes security flaws located in the program [42, 71, 72, 80, 91, 104, 126]. Moreover, pioneering researchers proposed multiple variant systems to prevent a successful exploit that leverages common vulnerabilities from being widespread across similar environments [35, 53]. These lines of work share the same purposes with DeView, increasing the level of difficulty for adversaries to understand a target program and craft an exploit that works for all other similar programs. However, DeView is distinct in terms of what to transform and how to transform. DeView varies web API support per PWA in the same browser binaries and diversifies browser libraries by eliminating unwanted web APIs rather than inserting or modifying a small piece of code in the libraries.

9 CONCLUSION

Securing PWAs is important as they have become widely used by many application developers for various target platforms. We propose DeView, which can practically confine a PWA by eliminating unnecessary web APIs from a corresponding browser instance. To this end, DeView adopts record-and-replay web API profiling and compiler-assisted on-demand browser instrumentation. Our experiments show that DeView eliminates 91.8% of the whole web APIs per application on average, ranging from 75.5% to 98.5% with real-world PWAs. Also, DeView can prevent 76.3% of CVEs (out of 478) relevant to web API exploits on average. We demonstrate the practicality of DeView with reasonable performance overheads.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Yinzhi Cao, for their constructive feedback. This work was supported by the Basic Science Research Program through NRF grant funded by the Ministry of Education of the Government of South Korea (No. 2022R1F1A107437311). Also, it was supported, in part, by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2022-0-01199; Graduate School of Convergence Security (SungKyunKwan university), No.2022-0-00688; AI Platform to Fully Adapt and Reflect Privacy-Policy Changes). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the sponsor's views.

REFERENCES

[1] Paul Adenot and Hongchan Choi. 2020. Web Audio API. https://www.w3.org/TR/webaudio/.

[2] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*. 70–83.

[3] Ali Alabbas and Joshua Bell. 2020. Indexed Database API 3.0. https://w3c.github.io/IndexedDB.

[4] Abdulrahman Alqabandi. 2020. Firefox uXSS and CSS XSS. https://leucosite.com/Firefox-uXSS-and-CSS-XSS/.

[5] Amazon. 2020. Top Sites in United States. https://www.alexa.com/topsites/countries/US.

[6] Appscope. 2020. Appscope. https://appsco.pe.

[7] Jake Archibald. 2019. The Service Worker Lifecycle. https://developers.google.com/web/fundamentals/primers/service-workers/lifecycle#install.

[8] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*.

[9] Vivek Balachandran and Sabu Emmanuel. 2013. Software protection with obfuscation and encryption. In *International Conference on Information Security Practice and Experience*. Springer, 309–320.

[10] Vivek Balachandran, Sabu Emmanuel, and Ng Wee Keong. 2014. Obfuscation by code fragmentation to evade reverse engineering. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 463–469.

[11] Kayce Basques. 2020. Find Unused JavaScript And CSS Code With The Coverage Tab In Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools/coverage.

[12] Mihai Bazon. 2020. UglifyJS. http://lisperator.net/uglifyjs.

[13] Bobby D Birrer, Richard A Raines, Rusty O Baldwin, Barry E Mullins, and Robert W Bennington. 2007. Program fragmentation as a metamorphic software protection. In *Third International Symposium on Information Assurance and Security*. IEEE, 369–374.

[14] Lauren Bradley. 2019. Starbucks and Ipsy Win with eCommerce PWA and SPA Frontends. https://www.layer0.co/post/starbucks-ipsy-win-ecommerce-progressive-web-apps-single-page-applications.

[15] The brave community. 2019. Spotify web player is not working any more. https://community.brave.com/t/spotify-web-player-is-not-working-any-more/75752/9.

[16] Jan Böhmer. 2020. Crooked Style Sheets. https://github.com/jbtronics/CrookedStyleSheets.

[17] Checkly. 2020. Headless Recorder. https://github.com/checkly/headless-recorder.

[18] Erik Chen. 2017. Consistent Memory Metrics in Task Manager. https://docs.google.com/document/d/1PZyRzChnvkUNUB85Op46aqkFXuAGUJi751DJuB6O40g.

[19] Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. DamGate: Dynamic Adaptive Multi-feature Gating in Program Binaries. In *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*.

[20] Seongje Cho, Hyeyoung Chang, and Yookun Cho. 2008. Implementation of an obfuscation tool for c/c++ source code protection on the xscale architecture. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 406–416.

[21] The chrome team. 2020. Chrome Releases. https://chromereleases.googleblog.com/.

[22] The chromium project. 2020. Chrome Platform Status. https://www.chromestatus.com/features.

[23] The chromium project. 2020. Issue 1065761: Security: Copy & paste XSS via noscript. https://bugs.chromium.org/p/chromium/issues/detail?id=1065761.

[24] The chromium project. 2020. MemoryInfra. https://chromium.googlesource.com/chromium/src/+/master/docs/memory-infra/README.md.

[25] The chromium project. 2020. Web IDL Interfaces. https://www.chromium.org/developers/web-idl-interfaces.

[26] Ian Clelland. 2020. Permissions Policy. https://www.w3.org/TR/permissions-policy-1/.

[27] Andy Cockburn, Saul Greenberg, Bruce McKenzie, Michael Jasonsmith, and Shaun Kaasten. 1999. WebView: A graphical aid for revisiting Web pages. In *Proceedings of the OZCHI Australian Conference on Human Computer Interaction*.

[28] The Mozilla community. 2020. WebIDL. https://firefox-source-docs.mozilla.org/dom/bindings/webidl/index.html.

[29] The Mozilla community. 2022. Diagnose Firefox issues using Troubleshoot Mode. https://support.mozilla.org/en-US/kb/diagnose-firefox-issues-using-troubleshoot-mode.

[30] The WebKit Community. 2017. WebKit IDL. https://trac.webkit.org/wiki/WebKitIDL.

[31] MDN contributors. 2020. Introduction to web APIs. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction.

[32] MDN contributors. 2020. Web APIs. https://developer.mozilla.org/en-US/docs/Web/API.

[33] Starbucks Corporation. 2020. Starbucks Coffee Company. https://app.starbucks.com.

[34] Rafael Costa, Luci Pirmez, Davidson Boccardo, Luiz Fernando Rust, and Raphael Machado. 2012. TinyObf: code obfuscation framework for wireless sensor networks. In *Proceedings International Conference on Wireless Networks (ICWN)*. 68–74.

[35] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. 2006. N-Variant Systems: A Secretless Framework for Security through Diversity.. In *Proceedings of the 15th USENIX Security Symposium (Security)*. Vancouver, Canada.

[36] Marcos Cáceres, Kenneth Rohde Christiansen, Mounir Lamouri, Anssi Kostiainen, Matt Giuca, and Aaron Gustafson. 2020. Web App Manifest. https://www.w3.org/TR/appmanifest.

[37] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. Sysfilter: Automated System Call Filtering for Commodity Software. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*.

[38] Google Developers. 2020. Puppeteer. https://developers.google.com/web/tools/puppeteer.

[39] Google Developers. 2022. Installation prompt. https://web.dev/learn/pwa/installation-prompt.

[40] Alexis Deveria. 2020. Permissions API. https://caniuse.com/permissions-api.

[41] OpenJS Foundation. 2020. Electron: Build cross-platform desktop apps with JavaScript, HTML, and CSS. https://www.electronjs.org.

[42] Kazuhide Fukushima, Shinsaku Kiyomoto, and Toshiaki Tanaka. 2009. Obfuscation mechanism in conjunction with tamper-proof module. In *2009 International Conference on Computational Science and Engineering*, Vol. 2. IEEE, 665–670.

[43] Masoud Ghaffarinia and Kevin W. Hamlen. 2019. Binary Control-Flow Trimming. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*.

[44] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*.

[45] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*.

[46] Nizamettin Gok and Nitin Khanna. 2013. *Building Hybrid Android Apps with Java and JavaScript*. O'Reilly Media, Inc.

[47] Reilly Grant, Ken Rockot, and Ovidio Ruiz-Henríquez. 2020. WebUSB API. https://wicg.github.io/webusb/.

[48] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2014. FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[49] Mike Gualtieri. 2018. Stealing Data With CSS: Attack and Defense. https://www.mike-gualtieri.com/posts/stealing-data-with-css-attack-and-defense.

[50] Philip J. Guo and Dawson Engler. 2011. CDE: Using System Call Interposition to Automatically Create Portable Software Packages.

[51] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*.

[52] Hemanth HM. 2020. Awesome PWA. https://github.com/hemanth/awesome-pwa.

[53] Kjell Jørgen Hole. 2013. Diversity reduces the impact of malware. (May 2013).

[54] E. Horton and C. Parnin. 2019. DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*.

[55] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. 2018. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology* 104 (2018), 72–93.

[56] Lin-Shung Huang, Alex Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. 2012. Clickjacking: Attacks and Defenses. In *Proceedings of the 21st USENIX Security Symposium (Security)*. Bellevue, WA.

[57] Tomohiro Ikeda. 2020. XSound. https://xsound.app/.

[58] Apple Inc. 2020. WKWebView. https://developer.apple.com/documentation/webkit/wkwebview.

[59] Absolute Markets Insights. 2019. Progressive Web Apps Market 2019 2027. https://www.absolutemarketsinsights.com/reports/Progressive-Web-Apps-Market-2019-2027-414.

[60] jcpazos. 2018. Some blocked features still accessible. https://github.com/pes10k/web-api-manager/issues/97.

[61] Cullen Jennings, Henrik Boström, and Jan-Ivar Bruaroey. 2020. WebRTC 1.0: Real-Time Communication Between Browsers. https://www.w3.org/TR/webrtc/.

[62] Jun. 2017. PWA - Progressive Web Attack. https://shhnjk.blogspot.com/2017/10/pwa-progressive-web-attack.html.

[63] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Hong Kong.

[64] Antoni Kepinski. 2019. How to make your Electron app faster. https://dev.to/xxczaki/how-to-make-your-electron-app-faster-4ifb.

[65] Kirupa. 2019. Understanding WebViews. https://www.kirupa.com/apps/webview.htm.

[66] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security (EuroSec)*.

[67] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2020. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*.

[68] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schroder-Preikschat, Daniel Lohmann, and Rudiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[69] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. 2018. Pride and Prejudice in Progressive Web Apps: Abusing Native App-like Features in Web Applications. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.

[70] Google LLC. 2020. WebView for Android. https://developer.chrome.com/multidevice/webview/overview.

[71] Kangjie Lu, Siyang Xiong, and Debin Gao. 2014. Ropsteg: program steganography with return oriented programming. In *Proceedings of the 4th ACM conference on Data and application security and privacy*. 265–272.

[72] Anirban Majumdar and Clark Thomborson. 2006. Manufacturing opaque predicates in distributed systems for code obfuscation. In *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*. Citeseer, 187–196.

[73] marmelab. 2020. gremlins.js. https://github.com/marmelab/gremlins.js.

[74] Cameron McCormack, Yves Lafon, and Travis Leithead. 2020. WebIDL Level 1. https://www.w3.org/TR/WebIDL/.

[75] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Toward Detecting andPreventing DOM Cross-Site Scripting. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[76] Jan M Memon, Asghar Mughal, Faisal Memon, et al. 2006. Preventing reverse engineering threat in Java using byte code obfuscation techniques. In *2006 International Conference on Emerging Technologies*. IEEE, 689–694.

[77] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*.

[78] Shachee Mishra and Michalis Polychronakis. 2020. Saffire: Context-sensitive Function Specialization against Code Reuse Attacks. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroS&P)*.

[79] Akito Monden, Antoine Monsifrot, and Clark Thomborson. 2003. Obfuscated instructions for software protection. *Information Science Technical Report, NAIST-IS-TR2003013, Nara Institute of Science and Technology* (2003).

[80] Akito Monden, Antoine Monsifrot, and Clark Thomborson. 2004. A framework for obfuscated interpretation. In *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation-Volume 32*. 7–16.

[81] Max Moroz and Sergei Glazunov. 2019. *Analysis of UXSS exploits and mitigations in Chromium*. Technical Report. Google LLC.

[82] Mozilla and individual contributors. 2020. Polyfill. https://developer.mozilla.org/en-US/docs/Glossary/Polyfill.

[83] Mozilla and individual contributors. 2020. Standard built-in objects. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects.

[84] Mozilla and individual contributors. 2020. Using Promises. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises.

[85] Mozilla and individual contributors. 2020. Vendor Prefix. https://developer.mozilla.org/en-US/docs/Glossary/Vendor_Prefix.

[86] Mozilla and individual contributors. 2022. BeforeInstallPromptEvent – Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/BeforeInstallPromptEvent.

[87] Mozilla and individual contributors. 2022. Scope – Web app manifests | MDN. https://developer.mozilla.org/en-US/docs/Web/Manifest/scope.

[88] Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking Payloads with Runtime Code Stripping and Image Freezing.

[89] neutrinos. 2021. Progressive Web Apps vs Native Apps. https://www.goneutrinos.com/wp-content/uploads/2021/06/Whitepaper-Progressive-Web-Apps-vs-Native-Apps.pdf.

[90] Pauli Olavi Ojala. 2017. Put your Electron app on a diet with Electrino. https://medium.com/dailyjs/put-your-electron-app-on-a-diet-with-electrino-c7ffdf1d6297.

[91] Rasha Omar, Ahmed El-Mahdy, and Erven Rohou. 2014. Arbitrary control-flow embedding into multiple threads for obfuscation: A preliminary complexity and performance analysis. In *Proceedings of the 2nd international workshop on Security in cloud computing*. 51–58.

[92] Addy Osmani. 2015. Getting started with Progressive Web Apps. https://developer.chrome.com/blog/getting-started-pwa/.

[93] Stack Overflow. 2021. 2021 Developer Survey. https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language-prof.

[94] Arthur Poot. 2020. The state of PWA support on mobile and desktop in 2020. https://simplabs.com/blog/2020/06/10/the-state-of-pwa-support-on-mobile-and-desktop-in-2020/.

[95] progressivewebapproom@gmail.com. 2020. Great examples of progressive web apps in one room. http://progressivewebapproom.com.

[96] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the 28th USENIX Security Symposium*.

[97] Chenxiong Qian, HyungJoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*. Virtual Event.

[98] Jiancheng Qin, Zhongying Bai, and Yuan Bai. 2008. Polymorphic algorithm of javascript code protection. In *2008 International Symposium on Computer Science and Computational Technology*, Vol. 1. IEEE, 451–454.

[99] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*. 869–886.

[100] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick D. McDaniel. 2017. Cimplifier: automatically debloating containers. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*.

[101] Sam Richard and Pete LePage. 2020. What makes a good Progressive Web App? https://web.dev/pwa-checklist/.

[102] Alex Russell, Jungkee Song, Jake Archibald, and Marijn Kruisselbrink. 2020. Service Workers Nightly. https://w3c.github.io/ServiceWorker/.

[103] Shaown Sarker, Jordan Jueckstock, and Alexandros Kapravelos. 2020. Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage. In *Proceedings of the 20th ACM Internet Measurement Conference (IMC)*. Pittsburgh, PA.

[104] Sebastian Schrittwieser and Stefan Katzenbeisser. 2011. Code obfuscation against static and dynamic reverse engineering. In *International workshop on information hiding*. Springer, 270–284.

[105] Liang Shan and Sabu Emmanuel. 2011. Mobile agent protection with self-modifying code. *Journal of Signal Processing Systems* 65, 1 (2011), 105–116.

[106] Ax Sharma. 2022. Dev corrupts NPM libs 'colors' and 'faker' breaking thousands of apps. https://www.bleepingcomputer.com/news/security/dev-corrupts-npm-libs-colors-and-faker-breaking-thousands-of-apps.

[107] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. 2016. Browser Feature Usage on the Modern Web. In *Proceedings of the 16th ACM Internet Measurement Conference (IMC)*. Los Angeles, CA.

[108] Peter Snyder, Cynthia Taylor, and Chris Kanich. 2017. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*.

[109] Cybersecurity Help s.r.o. 2019. Cross-site scripting in PWA for WP & AMP for WordPress. https://www.cybersecurity-help.cz/vdb/SB2019032518.

[110] Simon Stewart and David Burns. 2020. WebDriver. https://www.w3.org/TR/webdriver/.

[111] Naoki Takei, Takamichi Saito, Ko Takasu, and Tomotaka Yamada. 2015. Web Browser Fingerprinting Using Only Cascading Style Sheets. In *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*. 57–63.

[112] Telegram. 2020. Telegram Web. https://webweb.telegram.org.

[113] The terser community. 2020. JavaScript parser, mangler and compressor toolkit for ES6+. https://terser.org/.

[114] Dan Thorp-Lancaster. 2018. Microsoft announces Teams Progressive Web App (PWA) preview for Windows 10 S. https://www.windowscentral.com/microsoft-announces-teams-progressive-web-app-pwa-preview-windows-10-s.

[115] Common Vulnerabilities and Exposures. 2019. CVE-2019-13720. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13720.

[116] Common Vulnerabilities and Exposures. 2020. CVE-2020-6541. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-6541.

[117] Common Vulnerabilities and Exposures. 2021. CVE-2021-4079. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-4079.

[118] W3C. 2020. W3C Working Group. https://www.w3.org/groups/wg/.

[119] Tom Warren. 2019. Microsoft has turned Outlook into a Progressive Web App. https://www.theverge.com/2019/11/26/20983886/microsoft-outlook-com-

pwa-progressive-web-app-install-features.
[120] Wikipedia. 2020. INT (x86 instruction). https://en.wikipedia.org/wiki/INT_(x86_instruction).
[121] Wikipedia. 2020. Jaccard index. https://en.wikipedia.org/wiki/Jaccard_index).
[122] Wikipedia. 2022. Progressive web application. https://en.wikipedia.org/wiki/Progressive_web_application#Technologies.
[123] Wikipedia. 2022. WHATWG. https://en.wikipedia.org/wiki/WHATWG.
[124] Shucai Xiao, Yanzhu Ye, et al. 2009. Tamper resistance for software defined radio software. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, Vol. 1. IEEE, 383–391.
[125] Hongfa Xue, Yurong Chen, Guru Venkataramani, and Tian Lan. 2019. Hecate: Automated Customization of Program and Communication Features to Reduce Attack Surfaces. In *International Conference on Security and Privacy in Communication Systems (SecureComm)*.
[126] Ding Yi. 2009. A new obfuscation scheme in constructing fuzzy predicates. In *2009 WRI World Congress on Software Engineering*, Vol. 4. IEEE, 379–382.
[127] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What are weak links in the NPM supply chain?. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.
[128] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. 2018. KASR: A Reliable and Practical Approach to Attack Surface Reduction of Commodity OS Kernels. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*. 691–710.

A APPENDIX

A.1 Chromium Build Configuration

Configuration	Value	Configuration	Value
`is_component_build`	`true`	`symbol_level`	`0`
`is_chrome_branded`	`true`	`blink_symbol_level`	`0`
`is_debug`	`false`	`ffmpeg_branding`	`"Chrome"`
`enable_nacl`	`false`	`clang_base_path`	`/* Path to Clang */`
`properietary_codecs`	`true`		

Table 4: Chromium build configuration for DeView. The number of available web APIs may vary according to it.

A.2 Case Study: Starbucks PWA

In this section, we demonstrate the robustness and effectiveness of DeView with a real-world PWA, e.g., the commercial Starbucks PWA [33]. Note that we created a few test accounts to collect exercised web APIs after a sign-in. We begin with a profiling process on a local workstation using our instrumented Chromium. Unlike our assumption of a deployment model, we cannot directly deploy a required web API list along with `Web App Manifest` for the Starbucks PWA because we do not own its web server. Instead, we install the Starbucks PWA from an application server, record user activities (e.g., browsing), and then place them in the manifest directory. Next, we forcefully trigger a service worker update event via Chromium's DevTool and run our profiling tool to export the exercised web API list with replaying, followed by generating the debloated browser engine libraries. Note that all processes can be automatically completed except recording user behaviors. We confirm that the debloated version of the browser engine works flawlessly, successfully blocking four known proof-of-concept exploits, including XSS, URL spoofing, and two CSS attacks.

Protection against CVEs. We choose four proof-of-concept exploits to showcase the effectiveness of our approach: ① XSS, ② URL spoofing, and ③ two CSS-based attacks: fingerprinting and input field eavesdropping. First, `CVE-2018-6145` describes an XSS attack via `MathML` when Chromium's HTML parser does not appropriately handle a specially crafted string starting with a `<math>` tag. It unintentionally lets an adversary inject an arbitrary script. We ran a Chromium browser with and without the debloated browser engine libraries by loading a page containing the exploit separately. Since the Starbucks PWA needed neither Math nor XML tags, DeView disabled the corresponding web APIs in the debloated browser engine for it. We confirmed that our approach successfully protected the PWA from the exploit. Second, `CVE-2020-6431` can be leveraged in a URL spoofing attack that utilizes the fullscreen API (e.g., `requestFullScreen` and `webkitRequestFullscreen`). That is, attackers can display a fake website in fullscreen to lure users. Since the Starbucks PWA did not need the fullscreen API, the debloated browser engine also did not have the API, so the attack was unsuccessful. Third, CSS can be weaponized to exfiltrate sensitive information [111]. We examined the debloated browser engine with the crooked style sheets demo [16] and the CSS exfil attack [49]. The common idea of both CSS attacks is to conditionally load an external resource through a `URL` by abusing CSS selectors (e.g., HTML attribute selector, media query) and properties (e.g., `background`, `background-image`, `cursor`). Based on a request, an attacker can identify the victim or leak sensitive information. Since the two attacks rely on `background` or `content` that the Starbucks PWA does not require, they failed in the debloated browser engine. The attacker might circumvent our defense by replacing the removed CSS properties with the remaining alternatives, such as `background-image` or `cursor`. However, we emphasize that DeView still helps mitigate the implication of an exploit because it prevents the attacker from crafting a universal attack that works for every PWA by diversifying the browser engine (i.e., debloated variants). This case study demonstrates that DeView can effectively protect a real-world PWA from existing exploits or at least significantly raise the bar for attackers by constraining web APIs available.

A.3 Removable Web APIs for Real-World PWAs

PWA Name	Web APIs (8249)	Removed HTML (211)	Removed CSS (590)	Removed JS Object (837)	Removed JS API (7448)	Prevented CVEs (478)	PWA Name	Web APIs (8249)	Removed HTML (211)	Removed CSS (590)	Removed JS Object (837)	Removed JS API (7448)	Prevented CVEs (478)
AlarmDJ	1,219	78.20%	9.32%	86.74%	91.43%	338	Minesweeper	338	89.57%	78.98%	94.86%	97.42%	405
Alibaba	1,280	66.35%	71.02%	81.12%	86.06%	268	MoneyTracker	500	84.36%	75.59%	92.23%	95.66%	395
Aliexpress	1,068	64.93%	71.53%	83.27%	88.91%	290	MTGStocks	794	70.14%	72.71%	89.01%	92.35%	354
AMP	1,009	66.35%	41.36%	86.26%	92.05%	338	MultiCalc	536	82.46%	74.41%	92.11%	95.33%	388
Anonynote	828	78.20%	73.22%	86.98%	91.62%	346	MusicKit	350	89.57%	75.25%	94.50%	97.56%	419
ArrowsRain	126	93.84%	89.83%	97.97%	99.29%	441	Notepad	392	82.94%	77.97%	94.86%	96.97%	405
Avain	209	91.47%	84.58%	96.54%	98.66%	415	Pencil	613	79.15%	75.76%	90.68%	94.28%	361
BentoStarter	671	77.25%	71.86%	89.84%	93.86%	322	Pinterest	849	76.30%	72.37%	86.74%	91.46%	320
BestMarkdown-Editor	495	84.83%	77.97%	91.52%	95.53%	366	Pokedex	327	88.15%	77.12%	94.03%	97.76%	411
Booksie	607	79.15%	73.90%	89.96%	94.51%	358	PokeQuestWiki	523	79.15%	74.75%	91.88%	95.57%	384
BreakLock	220	86.73%	78.81%	97.49%	99.10%	440	PregBuddy	457	85.78%	78.81%	93.07%	95.95%	348
BubblePairs	302	87.20%	82.88%	95.94%	97.66%	414	ProgressiveBeer	540	80.57%	73.73%	90.32%	95.38%	393
BudgetTracker	1,031	78.67%	9.49%	89.25%	93.93%	319	PWAReact-Calculator	539	79.62%	76.44%	91.28%	95.21%	381
CareCards	676	71.56%	69.49%	89.73%	94.15%	357	PWAReact-MusicPlayer	648	70.14%	72.71%	90.20%	94.31%	331
ChromeDeveloper-Summit	1,410	61.14%	9.32%	84.35%	89.35%	326	QRCodeGenerator	431	88.63%	78.31%	92.83%	96.25%	404
chromestatus	1,472	66.82%	9.32%	81.60%	88.36%	312	QRCodeScanner	517	80.57%	75.59%	91.52%	95.54%	363
Closerintime	1,015	85.31%	9.49%	90.68%	93.96%	379	QRSnapper	256	91.00%	82.03%	96.06%	98.24%	413
Colosseum	778	87.68%	9.66%	93.79%	97.06%	392	ReactWeather	309	90.05%	81.53%	95.34%	97.60%	423
CurrencyConverter	846	61.14%	69.49%	88.05%	92.16%	336	Regretris	178	94.31%	91.86%	96.18%	98.42%	424
CurrencyExchange-LossCalculator	521	78.20%	75.76%	92.47%	95.54%	387	Remember	279	91.94%	78.81%	96.30%	98.16%	420
Datememe	950	66.35%	70.85%	84.47%	90.51%	302	RenzysYahtzee	319	87.20%	82.88%	94.74%	97.44%	408
DeadOrAlive	316	91.00%	83.05%	94.86%	97.35%	413	ResumeNation	1,037	75.36%	9.49%	90.56%	93.94%	362
DevOpera	619	71.09%	72.37%	91.04%	94.70%	365	SaintsSchedule	194	92.42%	91.86%	96.30%	98.25%	413
Dice	164	92.89%	85.25%	97.37%	99.17%	446	SantaTracker	949	80.57%	71.19%	84.95%	90.09%	340
Dino	207	93.84%	87.12%	96.06%	98.42%	435	SimilarWorlds	1,048	74.88%	71.69%	83.75%	88.88%	315
DoodleCricket	592	82.46%	73.90%	90.56%	94.62%	343	SimpleCurrency-Converter	412	92.89%	78.14%	92.59%	96.40%	405
ELFSH	328	83.41%	80.00%	95.46%	97.65%	411	Skcript	1,228	72.04%	9.49%	86.98%	91.47%	337
Emberclear	510	75.83%	73.56%	92.23%	95.93%	395	SmallerPictures	435	86.73%	77.97%	92.83%	96.28%	376
Emojityper	358	87.20%	81.02%	94.03%	97.06%	403	Snake	229	93.36%	91.86%	94.98%	97.76%	418
Encounters	684	74.41%	76.10%	89.37%	93.43%	360	Snapdrop	318	89.57%	78.81%	94.74%	97.70%	385
Etch	415	76.78%	77.29%	94.50%	96.89%	410	Soundslice	1,017	71.09%	67.46%	83.75%	89.74%	294
FinancialTimes	1,355	53.55%	66.10%	80.05%	85.81%	281	SplittyPie	650	77.25%	74.41%	89.25%	93.94%	378
FirefoxPlatformStatus	647	58.77%	68.98%	91.04%	94.94%	358	Starbucks	1,115	74.41%	70.68%	81.96%	88.08%	295
FlagWarriors	431	84.83%	81.53%	93.79%	96.11%	388	svginger	452	89.57%	81.53%	93.19%	95.69%	389
GitHubExplorer	311	89.10%	79.83%	95.34%	97.73%	410	SVGOMG	540	83.41%	74.07%	90.56%	95.27%	374
GlobalDefense	207	94.31%	85.25%	97.01%	98.55%	423	Telegram	894	71.56%	70.00%	85.07%	91.18%	310
GoogleMap	1,298	61.14%	67.12%	80.76%	86.28%	257	Tetra	251	93.36%	87.63%	95.10%	97.80%	417
GoogleNews	1,999	47.39%	8.47%	74.31%	81.90%	248	TheCircle	450	93.36%	67.63%	93.55%	97.44%	403
GooglePhotos	1,342	61.61%	52.37%	80.88%	86.84%	262	Themer	524	85.78%	73.90%	91.88%	95.44%	357
GrrdsTicTacToe	192	86.73%	82.54%	97.61%	99.18%	426	TheTrendBed	790	79.62%	73.73%	87.22%	92.05%	346
Grubhub	1,266	58.29%	67.63%	80.65%	86.75%	286	TicTacToe	381	91.47%	84.92%	93.43%	96.32%	401
GuitarTuner	314	91.00%	90.00%	93.43%	96.83%	393	Tinder	1,162	73.93%	41.36%	84.59%	89.78%	278
HackerNews	662	72.99%	72.03%	89.96%	94.09%	338	TotalFormatter	509	86.26%	74.75%	92.71%	95.56%	379
iHeartRadio	2,018	63.51%	9.32%	75.27%	81.12%	231	TowerGame	254	92.89%	88.47%	95.34%	97.70%	413
Indecisive	121	92.89%	86.78%	98.57%	99.62%	446	trivago	1,411	59.72%	41.36%	80.88%	86.84%	290
journalistic	470	81.04%	76.44%	92.83%	96.09%	398	Twitter	1,166	67.30%	68.47%	81.72%	87.77%	274
jsfeatures	233	90.05%	84.07%	96.77%	98.42%	426	Uber	1,175	72.99%	66.27%	84.11%	87.66%	311
JSONFormatter	293	91.47%	82.03%	95.58%	97.73%	438	Unsplash	1,239	52.13%	63.90%	81.48%	87.58%	303
Kahla	522	78.67%	75.59%	91.64%	95.53%	393	VaporBoy	612	82.46%	74.41%	89.49%	94.31%	316
KlondikeSolitaire	240	85.31%	78.14%	97.13%	98.93%	430	Versus	1,788	67.30%	9.32%	79.21%	84.10%	269
Letgo	1,436	62.09%	67.97%	77.06%	84.33%	266	Wavemaker	871	73.93%	73.05%	85.54%	91.18%	303
LofiNews	354	89.57%	81.53%	94.38%	97.01%	404	WavePD1	289	89.57%	83.22%	94.62%	97.74%	423
MakeBetterSoftware	614	82.46%	75.42%	90.08%	94.20%	365	WeatherApp	349	94.31%	80.17%	93.79%	97.05%	427
MakeMyTrip	1,014	69.67%	73.73%	83.51%	89.33%	298	WebNFCEnabled-ShoppingCart	396	86.73%	78.47%	93.55%	96.76%	394
Mandala3D	286	92.42%	86.44%	95.46%	97.45%	416	XSound	607	83.41%	78.64%	89.37%	94.01%	366
MaskableApp	427	66.82%	73.05%	94.03%	97.34%	397	YouTubeMusic	1,436	73.46%	8.47%	83.99%	88.72%	286
MemoryGame	204	92.89%	85.42%	96.89%	98.62%	428	Yummly	1,832	65.88%	9.32%	78.38%	83.55%	234

Table 5: Results of real-world Progressive Web Applications that we have tested in alphabetical order (114 in total). The empirical results show that DeView can remove 79.75%, 68.25%, 90.24% and 94.04% on average for HTML, CSS, JS Object and JS API, respectively.

A.4 Preventable CVEs

Feature	Category	# CVEs	H/C/J	Feature	Category	# CVEs	H/C/J	Feature	Category	# CVEs	H/C/J
Accessibility	UaF	2	0/0/2	Fullscreen	Spoof	13	0/0/13	Stream	OOB Read	2	0/0/2
Animation	Bypass	1	0/0/1		UaF	2	0/0/2	SVG	Bypass	2	2/0/0
	UaF	2	0/0/2	Graphics	Overflow	2	1/0/1		Memory Corruption	1	0/0/1
Autofill	Bypass	4	2/1/1		UaF	2	0/0/2		OOB Write	1	1/0/0
	Disclosure	8	1/1/6	History	Bypass	2	0/0/2	TextDecoder	Overflow	1	0/0/1
	Overflow	2	2/0/0		Overflow	1	0/0/1	UI	Bypass	1	1/0/0
	Spoof	3	1/0/2		Spoof	1	0/0/1		Overflow	7	0/0/7
	UaF	3	1/0/2	ImageCapture	UaF	1	0/0/1		Spoof	8	2/0/6
Blob	Bypass	1	0/0/1	IndexedDB	Bypass	1	0/0/1		UaF	9	1/0/8
	Overflow	1	0/0/1		UaF	3	0/0/3		XSS	1	0/0/1
	UaF	1	0/0/1	Internal	Bypass	2	0/0/2	URL	Spoof	6	0/0/6
	XSS	1	0/0/1		Memory Corruption	2	0/0/2		XSS	1	0/0/1
Cache	Disclosure	1	0/0/1		OOB Write	1	0/0/1	WebAudio	Bypass	3	0/0/3
	UaF	2	0/0/2		Overflow	5	0/0/5		Disclosure	1	1/0/0
Canvas2D	Bypass	2	0/0/2		UaF	11	1/0/10		Memory Corruption	1	0/0/1
	Disclosure	3	0/0/3	JavaScript	Memory Corruption	1	0/0/1		OOB Read	3	0/0/3
	Memory Corruption	1	0/0/1		UaF	2	0/0/2		OOB Write	1	0/0/1
	Overflow	1	0/0/1	Loader	Disclosure	2	0/0/2		Overflow	2	0/0/2
	UaF	3	0/0/3		RCE	2	1/0/1		UaF	14	0/0/14
Contacts	Spoof	1	0/0/1		UaF	3	0/0/3	WebAuthn	UaF	1	0/0/1
Cookie	Bypass	1	0/0/1	MathML	XSS	1	1/0/0	WebCodecs	Memory Corruption	1	0/0/1
CSP	Bypass	18	2/0/16	Media	Bypass	2	0/0/2	WebGL	Disclosure	1	0/0/1
	Disclosure	2	1/0/1		Disclosure	3	0/0/3		Memory Corruption	2	0/0/2
	Overflow	1	0/0/1		Overflow	2	1/0/1		OOB Read	4	0/0/4
CSS	Disclosure	3	0/3/0		RCE	1	0/0/1		OOB Write	2	0/0/2
	Memory Corruption	1	0/1/0		Spoof	1	0/0/1		Overflow	20	0/0/20
	Spoof	1	0/1/0		UaF	8	0/0/8		Privilege Escalation	1	0/0/1
	UaF	3	0/2/1	Navigation	Bypass	8	1/0/7		UaF	8	0/0/8
DevTools	Disclosure	4	0/0/4		Disclosure	1	0/0/1	WebGPU	UaF	1	0/0/1
	Memory Corruption	1	0/0/1		Overflow	1	0/0/1	WebMIDI	UaF	1	0/0/1
	Privilege Escalation	1	0/0/1		Privilege Escalation	1	0/0/1	WebOTP	Bypass	1	0/0/1
	RCE	2	0/0/2		Spoof	15	1/0/14	WebRTC	Disclosure	2	0/0/2
	UaF	2	0/0/2		XSS	1	0/0/1		Memory Corruption	2	0/0/2
	XSS	1	0/0/1	Network	Disclosure	2	0/0/2		OOB Read	2	0/0/2
DOM	Bypass	3	1/0/2		Memory Corruption	1	0/0/1		OOB Write	1	0/0/1
	Disclosure	2	1/0/1		Overflow	1	0/0/1		Overflow	2	0/0/2
	Memory Corruption	2	1/0/1		UaF	2	0/0/2		UaF	6	0/0/6
	OOB Read	1	0/0/1	Password	UaF	3	0/0/3	WebSerial	OOB Read	1	0/0/1
	Overflow	1	0/0/1	Payment	Bypass	1	0/0/1	WebShare	Bypass	1	0/0/1
	Spoof	1	0/0/1		Disclosure	1	0/0/1		UaF	1	0/0/1
	UaF	5	3/0/2		UaF	9	0/0/9	WebSocket	Bypass	1	0/0/1
	XSS	1	1/0/0		XSS	1	0/0/1		Memory Corruption	1	0/0/1
Download	Bypass	6	1/0/5	PDF	Overflow	1	0/0/1	WebSpeech	Spoof	1	0/0/1
	Disclosure	1	1/0/0		UaF	3	2/0/1		UaF	4	0/0/4
	Spoof	3	0/0/3	Performance	Disclosure	8	0/0/8	WebSQL	OOB Read	1	0/0/1
	UaF	1	0/0/1	Presentation	RCE	1	0/0/1		Overflow	1	0/0/1
Editing	XSS	2	1/1/0		UaF	1	0/0/1		UaF	2	0/0/2
Extension	Bypass	5	0/0/5	Printing	Overflow	1	0/0/1	WebUSB	Overflow	1	0/0/1
	Disclosure	2	1/0/1		UaF	5	0/0/5		Spoof	2	0/0/2
	Memory Corruption	1	0/0/1	ReaderMode	UaF	1	0/0/1		UaF	2	0/0/2
	Privilege Escalation	3	0/0/3	Sandbox	Bypass	16	6/0/10	WebVideo	Bypass	1	1/0/0
	Spoof	1	0/0/1		Disclosure	2	0/0/2		Overflow	1	0/0/1
	UaF	3	0/0/3		Privilege Escalation	1	0/0/1		Spoof	1	1/0/0
Fetch	Disclosure	1	0/0/1		Spoof	1	1/0/0		UaF	1	0/0/1
FileSystem	Spoof	2	0/0/2		XSS	2	1/0/1	WebXR	UaF	2	0/0/2
	UaF	2	0/0/2	Sensor	UaF	2	0/0/2	Worker	Bypass	5	0/0/5
Font	Overflow	1	0/0/1	SOP	Bypass	1	0/0/1		Disclosure	4	0/0/4
	UaF	1	0/0/1		Disclosure	1	0/0/1		UaF	2	0/0/2
Form	Privilege Escalation	1	0/0/1	Storage	Disclosure	4	0/0/4	XHR	Disclosure	1	0/0/1
	UaF	1	0/0/1		OOB Read	1	0/0/1				
FTP	Disclosure	1	0/0/1		UaF	3	0/0/3				

Table 6: List of 478 CVEs pertaining to Chromium for the last five years (Jan. 2017 – Apr. 2022). We follow each vulnerability's feature and category information from `https://bugs.chromium.org` (i.e., component, description). H, C, and J represent HTML, CSS and JavaScript, respectively.