SLIMIUM: Debloating the Chromium Browser with Feature Subsetting

Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee

Georgia Institute of Technology

ABSTRACT

Today, a web browser plays a crucial role in offering a broad spectrum of web experiences. The most popular browser, Chromium, has become an extremely complex application to meet ever-increasing user demands, exposing unavoidably large attack vectors due to its large code base. Code debloating attracts attention as a means of reducing such a potential attack surface by eliminating unused code. However, it is very challenging to perform sophisticated code removal without breaking needed functionalities because Chromium operates on a large number of closely connected and complex components, such as a renderer and JavaScript engine. In this paper, we present SLIMIUM, a debloating framework for a browser (i.e., Chromium) that harnesses a hybrid approach for a fast and reliable binary instrumentation. The main idea behind SLIMIUM is to determine a set of *features as a debloating unit* on top of a hybrid (i.e., static, dynamic, heuristic) analysis, and then leverage *feature subsetting* to code debloating. It aids in i) focusing on security-oriented features, ii) discarding unneeded code simply without complications, and iii) reasonably addressing a non-deterministic path problem raised from code complexity. To this end, we generate a feature-code map with a *relation vector* technique and *prompt webpage profiling* results. Our experimental results demonstrate the practicality and feasibility of SLIMIUM for 40 popular websites, as on average it removes 94 CVEs (61.4%) by cutting down 23.85 MB code (53.1%) from defined features (21.7% of the whole) in Chromium.

CCS CONCEPTS

• **Security and privacy** → System security; Browser security.

KEYWORDS

Debloating; Browser; Program Analysis; Binary Rewriting

ACM Reference Format:

Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. SLIMIUM: Debloating the Chromium Browser with Feature Subsetting. In *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3372297.3417866

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417866

1 INTRODUCTION

Today, a web browser plays arguably the most important role to interface with a wide range of the Internet experiences from information searching, email checking and on-demand streaming to e-commerce activities. Moreover, the number of smart mobile users has been skyrocketing over the past decades, reaching up to 5.1 billion [48] in the world; that is, billions of people are considered potentially active users of web browsers to use various services. Of all, the Chrome browser [25] has dominated the market (around 65%) in both desktop and mobile environments since its release [1]. Unlike the initial design of Chromium, which aims to be a lightweight browser [36], its volume has been continuously inflated to meet numerous users' demands. Chromium includes a large number of third-party software (around 40% of source code), as a feature-rich application predominately relies on well-designed external libraries and components. As a result, Chromium has become an extremely complex application to support a broad spectrum of features (e.g., PDF viewer, real time communication or virtual reality), which keeps expanding with new requirements.

While an abundance of features in Chromium offers unprecedented web experiences, a large code base often brings about an unwelcome outcome from a security perspective because it inevitably exposes extensive attack vectors that an adversary attempts to compromise. Besides, it would be exacerbated when code dependencies are common; external code may often introduce a known vulnerability unless a patch is applied in a timely manner. Indeed, the security community has reported various attacks [1, 33] as well as countless bugs [9, 18]. Recently, Permissions Policy [46] (aka., Feature Policy) was introduced to handle myriad features; however, it is yet rudimentary rather than comprehensive or standardized. Snyder et al. [42] propose a browser extension that selectively blocks low-benefit and high-risk features with cost-benefit evaluation, applying each website to feature restriction. Although this approach successfully blocks 15 (out of the 74) Web API standards and avoids 52% of all CVEs without affecting the usability of 94.7% of the tested websites, the binary code for a feature implementation still resides in memory. The approach can be circumvented [43] because its hardening mechanism mainly lies in disabling features by intercepting JavaScript APIs.

To remedy this problem, code debloating is another emerging means to reduce such an attack surface by eliminating unneeded code. The main challenge arises from sophisticated removal while preserving needed code at all times. Prior works [15, 23, 28, 31, 38, 53, 55] largely rely on binary analysis to create a customized version. PieceWise [40] utilizes additional information from a compiler toolchain to generate a specialized library with higher precision.

[1] Strictly speaking, Chromium forms a basis for Chrome as an open-source project, which lacks proprietary features. In this paper, we use Chromium.

Meanwhile, recent works [11, 16, 51] leverage machine learning techniques for debloating.

However, none of the above approaches scales to a massive application such as a web browser. The code chunk in an official Chromium release is around 110MB, even larger than the Linux kernel. The primary factor in hindering web browser debloating originates from not only its volume but also a unique property: embedding many large inner-components that are strongly connected to each other. For example, the renderer (i.e., `Blink` in Chromium) goes through a very complex process to complete a requested page view from a DOM (Document Object Model or Web APIs) tree [20], making internal use of various components such as Skia and GPU. Similarly, the V8 JS engine exposes the Web APIs with a number of binding interfaces to the actual implementation. Indeed, our call graph from static analysis empirically shows that 86.7% functions (of all 483K nodes) are connected either directly or indirectly. A strong connection among these underlying components renders either static or dynamic analysis (even in a combined manner) impractical because it results in either little removable code from a dependency (i.e., call or control flow) graph or accidental code elimination from failing to trace drastically divergent paths even when visiting the same page.

In this paper, we present Slimium, a debloating framework for a browser such as Chromium, harnessing *features as a debloating unit* atop hybrid (i.e., static, dynamic, heuristic) techniques and leveraging them for fast and reliable binary instrumentation with *feature subsetting*. We choose open-source Chromium because of its popularity and diverse applications; further, we believe the impact would be non-negligible, taking three billion end users into account. The "unit features for debloating" gain several advantages: i) focusing on features pertaining to security, ii) removing unneeded code handily, and iii) addressing a non-deterministic path issue from code complexity. To the best of our knowledge, this is the first work that achieves successful software debloating on a Chromium scale.

Slimium consists of the following three parts: i) feature-code mapping generation, ii) website profiling, and iii) binary instrumentation based on those analyses. As a first phase, we determine a set of 164 Chromium features as attack surface vectors with thorough analysis on both the Web specification standards and the latest 364 CVEs for the last two years. We discovered 153 CVEs (42%) associated with 42 features in our set, which could potentially be removed. Next, we create a *feature-code map* in a semi-automated fashion, starting with a manual exploration on source code corresponding to predefined features and then discovering more binary functions pertaining to the features automatically. To this end, we devise a new heuristic means, dubbed a *relation vector* technique, enabling us to deduce more function candidates for a feature. Once the feature-code map is complete, we perform *prompt website profiling* to obtain non-deterministic code paths by visiting popular websites as a baseline to avoid accidental code elimination. Note that the above phases that generate supplementary information are a one-time processing for further debloating. Last, we produce a slim Chromium version for target websites of our interest based on the above artifacts. Our experimental results show that Slimium could successfully i) create an accurate feature-code map, ii) remove unneeded features while preserving needed code successfully, and iii) generate debloated versions that work flawlessly for ordinary browsing of the target websites.

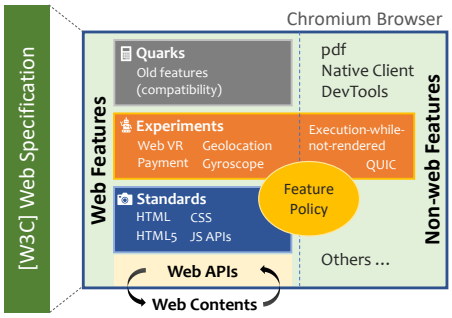

Figure 1: Web specification in Chromium.

In summary, we make the following contributions:

- We propose Slimium, a novel debloating framework for Chromium that allows one to access a pre-defined set of websites. Notably, we introduce a practical approach, feature subsetting, based on feature-code mapping with a relation vector and webpage profiling.
- We define a set of unit features for debloating. We thoroughly investigate both source code and the underlying components of Chromium to obtain the final set of removable features. To this end, we collect and analyze comprehensive CVEs that were officially fixed for the last two years.
- We evaluate the feasibility of our approach with a prototype Slimium implementation. The prototype demonstrates its effectiveness (reducing 61.4% of CVEs and 53.1% of code) and efficiency (less than a second to generate a debloated version) with high reliability (all Chromium variants work flawlessly).
- We provide an open source implementation for Slimium to foster debloating-relevant research, which can be obtained through https://github.com/cxreet/chromium-debloating.

2 BACKGROUND

In this section, we discuss a variety of Web standard specifications with important Web jargon to avoid further confusion, and the Chromium browser.

2.1 Browser Features and Web APIs

The World Wide Web Consortium (W3C), the international standard organization for the Web, has specified more than 1,200 standards [47] thus far. It is crucial to understand underlying components of a modern Web browser to determine the scope of unused code.

Figure 1 illustrates how Web specification has been deployed within a Chromium browser at a glance [2]. In general, a browser implements a subset of standard Web features. Non-Web features are browser-specific (The browser may include its own experimental features that could be standards later on). For instance, Native

[2] When the Web technology was immature, there was no clear boundary between Web and non-Web features because Web standards did not rule a Web browser. Here we assume a modern browser conforming to Web standards.

Client (NaCl) allows for secure execution of native code in a sandbox environment, which is only available in Chromium.

The web features can be categorized into three domains: `quarks`, `experiments`, and `standards`. A browser supports even deprecated features in a `quarks` set for backward compatibility. An `experiments` set defines a series of candidate features that come with mock implementation, which might be part of the `standards` specifications in the future. Chromium offers a list of flags [3] to activate various experimental features (i.e., QUIC protocol) for advanced users. A standards set implements all other specifications such as HTML, CSS, HTML5, and JavaScript APIs.

Web APIs play a pivotal role in interacting with the above feature sets. For example, a developer can utilize the `Geolocation` API when a functionality of geographical location is needed upon a user's approval. The Web APIs are typically exposed as JavaScript interfaces. Hence, in this paper, one of the attack vectors includes a browser feature defined as a JS object, method, or property. Indeed, JS APIs are a major entry point that has been weaponized by attackers [1, 18, 33, 50]. From an implementation view, Chromium internally specifies Web APIs through Web Interface Definition Language (IDL). `Blink` [37] has a dialect of Web IDL for binding actual implementation with each Web API in Chromium.

2.2 Feature Policy

Although ever-growing web features enrich users' experiences, the attack surface of web applications gets larger accordingly. A concept of *Feature Policy* [46] has been introduced, which allows a web server to selectively enable or disable a specific feature of a user agent. Similar to Content Security Policy (CSP) [29], Feature Policy defines a set of web-feature-relevant policies that restricts a behavior of the browser via the "Feature-Policy" HTTP header field (i.e., the server sends policy directives to the client for policy enforcement). We found around 30 feature directives in MDN [30] and Chromium documents [14] for Feature Policy, yet no directive has been officially standardized. The Chromium version for our experiment supports 25 feature directives [4].

However, we decide to define a new feature set that assists debloating for the following reasons: i) the standard features supported by Feature Policy are neither comprehensive nor implemented across browsers yet, ii) some features do not fit well on debloating because they may span multiple features. As an example, a `camera` feature implementation is part of both `WebRTC` and `Media Stream`. In this scenario, eliminating `camera` affects the other two features, rendering a debloating process opaque, and iii) it does not cover complete code that accounts for unneeded features to be potentially removed (e.g., PDF).

2.3 The Chromium Browser

Since the first release of Google Chromium in 2008, its market share has reached 68.8% for desktops and 63.6% across all platforms in the world [44] by 2018. Besides, many browsers based on Chromium have been launched (*e.g.*, Brave, Vivaldi, Opera, Iron, etc.), incorporating customized features or concepts. Lately, the `Edge` browser has been migrated into a Chromium's V8 engine and a `blink` renderer from `Chakra` and `EdgeHTML`, respectively [54]. `Electron` [10] is a runtime framework to build cross-platform applications on top of Node.js and Chromium.

The proliferation of Chromium and its applications inevitably becomes a fruitful attack vector to an adversary, taking advantage of a large attack surface. While it is essential to shrink unneeded functionalities, to the best of our knowledge, no work has achieved successful debloating in Chromium.

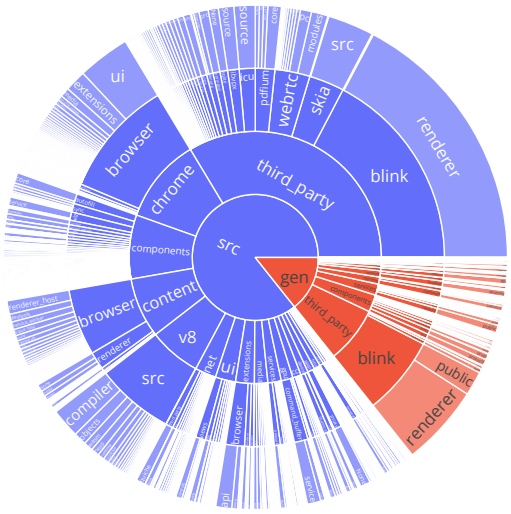

Figure 2: Hierarchical structure (a depth of four) of Chromium source directory: each portion represents an area in proportion to the actual size of (compiled) binary functions defined in an individual Chromium source path. Sub-directories are sorted in a counterclockwise direction.

2.4 Chromium Binary Structure

Chromium consists of a large chunk of code. The latest distributed version ships with approximately 110MB binary code. The main binary comprises around 23K compilation units (CUs) (i.e., object files), and 483K binary functions in total.

Figure 2 illustrates the entire directory structure of Chromium CUs at build time. Each fan-shaped area conveys the following: a) a hierarchical location (i.e., parent and child(ren)), b) the rate of the sum of binary function sizes within, and c) the rank of the rate in its parent (i.e., inner circle). For example, the `webrtc` resides in the `src/third_party/webrtc` whose function size takes up the third largest component in its parent directory, `third_party`. We leverage the structure information to choose feature candidates that could be eliminated (See Section 4.1).

The `gen` directory contains all source files generated at compilation time, which occupies a non-negligible portion (16.7%) of the Chromium codebase. A large amount of JavaScript V8 engine implementation comes from a script that creates binding code using a template as specified in a `Blink` IDL, which emits to the designated

[3] Entering `chrome://flags` at an address bar in Chromium gives an opportunity to enable experimental features ahead.
[4] With `document.featurePolicy.allowedFeatures()`, it returns features available in the current version of Chromium.

binding directory [5]. This means a single feature implementation may span multiple locations.

3 DEBLOATING CHROMIUM

In this section, we describe challenges and our approach along with Chromium debloating, and provide a SLIMIUM overview, followed by defining an attack surface for Chromium.

3.1 Challenges and Approach

At first glance, a code-slimming process sounds straightforward because its gist simply lies in identifying and nullifying unneeded code. However, on a large scale like Chromium, it is challenging to determine removable code because of not only its volume but also its inter-dependencies and interactions (e.g., inter-process communication, cache, asynchronization) between underlying components that are individually large enough. For example, the `Blink` renderer requires very expensive computations to present a web page view by building a DOM tree and drawing each pixel in a browser [20]. This process involves diverse internal components such as a parser, layout builder, painter, and compositor. Likewise, the bindings of Web APIs exposed by the V8 JS engine interface with many internal implementations (specified as IDL). These strongly connected components make static or dynamic analysis less useful (even in a hybrid manner) because i) using a call graph or control graph ends up with very little code that can be removed; e.g., the whole call graph constructed from our experiment contains 483K nodes and 1.5M edges, in which 86.7% of the nodes are connected to each other, and ii) exercised functions drastically differ from every visit of the identical webpage due to inherently non-deterministic behaviors (i.e., network, caching) or dynamic web contents (i.e., advertisements), as well as user interactions (i.e., keyboard and mouse events), resulting in accidentally eliminating needed code.

To address the above issues, we define *a feature as a debloating unit* and then *subset* the features for specific targets of our interest (See Section 4.1). The light-red box in Figure 3 presents our debloating approach at the high level. The features space (white squares) determines an attack surface to be potentially removed, whereas all other code is undebloatable. In this example, there are two features (F_1 and F_2) that consist of nine and 12 functions, respectively (a circle in each feature represents a function). With a baseline profiling result generated by visiting the Top 1000 Alexa websites, we mark non-deterministically exercised functions, that is, three functions for F_2 in (a). Next, we collect all exercised functions for a target page (or multiple pages) as (b), in which four functions have been additionally exercised for F_1. In this setting, we sorely eliminate nine functions in F_2 because the target page(s) adopts F_1, not F_2 as in (c). The three functions in F_2 stay intact, as they are possibly employed by other code non-deterministically. Note that the decision on whether a feature has been adopted would remain as a threshold because it strikes a balance between the size of possible code reduction and generation of a reliable variant.

3.2 SLIMIUM Overview

Figure 3 shows an overview of SLIMIUM for debloating Chromium. SLIMIUM consists of three main phases: i) feature-code mapping generation, ii) prompt website profiling based on page visits, and iii) binary instrumentation based on i) and ii).

Feature-Code Mapping. To build a set of unit features for debloating, we investigate source code [35] (Figure 2), previously-assigned CVEs pertaining to Chromium, and external resources [8, 47] for the Web specification standards (Step ❶ in Figure 3). Table 1 summarizes 164 features with four different categories. Once the features have been prepared, we generate a *feature-code map* that aids further debloating from the two sources (❶' and ❷'). From the light-green box in Figure 3, consider the binary that contains two CUs to which three and four consecutive binary functions (i.e., $\{f_0 - f_2\}$ and $\{f_3 - f_6\}$) belong, respectively. The initial mapping between a feature and source code relies on a manual discovery process that may miss some binary functions (i.e., from the source generated at compilation). Then, we apply a new means to explore such missing functions, followed by creating a call graph on the IR (Intermediate Representation) (Step ❷, Section 4.2).

Website Profiling. The light-yellow box in Figure 3 enables us to trace exercised functions when running a Chromium process. SLIMIUM harnesses a website profiling to collect non-deterministic code paths, which helps to avoid accidental code elimination. As a baseline, we perform differential analysis on exercised functions by visiting a set of websites (Top 1000 from Alexa [3]) multiple times (Step ❸). For example, we mark any function non-deterministic if a certain function is not exercised for the first visit but is exercised for the next visit. Then, we gather exercised functions for target websites of our interest with a defined set of user activities (Step ❹). During this process, profiling may identify a small number of exercised functions that belong to an unused feature (i.e., initialization). As a result, we obtain the final profiling results that assist binary instrumentation (❸' and ❹').

Binary Rewriting. The final process creates a debloated version of a Chromium binary with a feature subset (Step ❺ in Figure 3). In this scenario, the feature in the green box has not been needed based on the feature-code mapping and profiling results, erasing the functions $\{f_0, f_1, f_3\}$ of the feature. As an end user, it is sufficient to take Step ❹ and ❺ for binary instrumentation where pre-computed feature-code mapping and profiling results are given as supplementary information.

3.3 Chromium Attack Surface

A concept of an attack surface, in general, encompasses interfaces, protocols, address spaces or code itself, as a security measurement. Of our interest, the attack surface is a set of buggy code to be potentially misused in memory.

In this paper, we define the Chromium attack surface as the exposed code possibly leveraged by an adversary; that is, not all code is the target that could be weaponized. In the same vein, we exclude static HTML and CSS in our feature set because there is little margin for the adversary to leverage them to build a functional payload without the help of any JavaScript or HTML5 functionality. In this regard, our approach aims to neither achieve complete elimination of all unwanted features nor maximize the size of code reduction. Instead, our approach attempts to reduce a recognizable attack surface as the best-effort service.

[5] `gen/third_party/blink/renderer/bindings/*/v8`

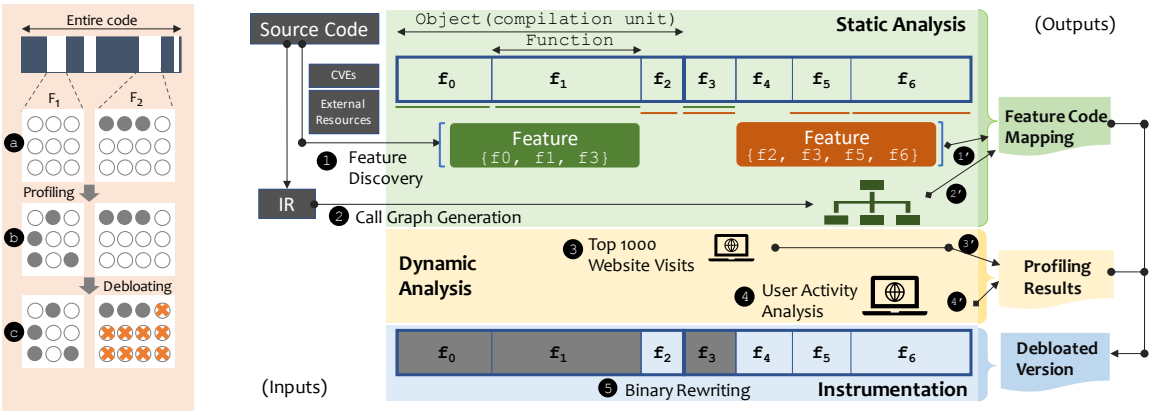

Figure 3: High-level overview of SLIMIUM. It leverages a concept of feature subsetting (feature as a unit of debloating) to guide a binary instrumentation as feedback on top of feature-code mapping and profiling results.

Table 1: Chromium features as a debloating unit (#: count).

Class	Features (#)	Functions (#)	Function Size (KB)	CVEs (#)	Feature Policy Directives (#)	Experimental Flags (#)
HTML5	6	8,103	1,721	15	0	0
JS API	100	71,082	17,204	57	25	15
Non-web	57	62,594	21,303	77	0	0
Wasm	1	1,189	869	4	0	0
Total	164	142,968	41,097	153	25	15

4 SLIMIUM DESIGN

In this section, we describe the design of SLIMIUM in detail.

4.1 Feature Set for Chromium Debloating

We begin with investigating all Web APIs to group them into different features from the approach in Snyder et al. [42], and exploring the Chromium's source code structure to include other features with an absence of Web APIs. Besides, we utilize external resources [8, 47] that list comprehensive features to define our final feature set for debloating. Note that we have excluded i) glue code that is commonly shared among multiple features and ii) code pertaining to fundamental security mechanisms such as SOP (Same Origin Policy) and CSP (Content Security Policy), which means that these security relevant features will be always retained in a debloated version of Chromium regardless of our profiling phase (4.3). In Table 1, we define 164 Chromium features that can be harnessed as a debloating unit, classifying them into four categories: JS API, HTML5, Non-web, and Wasm. Note that `wasm` (Web assembly) is the only feature that does not belong to HTML5, JS API, or the standard Web specifications. Interested readers can find further details regarding unit features in the Appendix (Table 4). In summary, a few notable statistical values are as follows: i) 153 CVEs reside in 42 debloatble features (25% of all the features), ii) 25 Feature Policy directives are included as part of our feature set, iii) 15 features can be enabled with an experimental flag, eight of which are defined as Feature Policy directives [6].

[6] accelerometer, ambient-light-sensor, fullscreen, magnetometer, gyroscope, vr, publickey-credentials, and xr-spatial-tracking

JavaScript API. As shown in Section 2.1, Chromium offers Web APIs that interact with web contents through JavaScript interfaces. In particular, we utilize `caniuse` [8] to classify the JavaScript APIs because it actively keeps track of browser-and-version-specific features as collective intelligence. Some of them have been combined due to a common implementation (i.e., `Blob` constructing and `Blob` URLs as a `Blob` API), resulting in 100 sub-categories.

HTML5. As the latest HTML version, HTML5 defines a rich feature set including audio, video, vector graphics (i.e., SVG, canvas), MathML, and various form controls as part of HTML by default. We define six major features that cause either a large code base or previous vulnerabilities (i.e., known CVEs). Recently, MarioNet [33] has demonstrated a new class of attacks that sorely relies on HTML5 APIs (i.e., a feature of service workers) in modern browsers, leading successfully unwanted operations.

Non-web Features. Our finding from Figure 2 shows that there are a few Chromium-browser-specific features such as devTools, extensions, and PDF that have been exposed to various attacks in the past years (Table 5). To exemplify, we could find 26 CVEs pertaining to a PDF feature alone. Additionally, we define each third-party component as a feature, assuming external code has a minimal dependency on each other. Indeed, this assumption holds for our features because their core implementations are often mutually exclusive. For example, few call invocations have been discovered among each other under the `third_party` directory based on our call graph analysis. Note that we have excluded a few of them when the feature is heavily employed by other parts such as `protobuf`.

4.2 Feature-Code Mapping

Generating a feature-code map is a key enabler to make our debloating approach feasible. In this section, we describe how to create such mapping in a reliable and efficient manner. To this end, we introduce a concept of a relation vector to seek more relevant code for a certain feature.

4.2.1 Manual Feature-Code Discovery. To determine the corresponding code to each feature, we begin with a manual investigation on source files that implement a certain feature, which is

worthwhile because Chromium often offers well-structured directories and/or file names and test suites. For example, the test set of the `battery` feature resides in `external/wpt/battery-status` and `battery-status` under the directory of `blink/web_tests` that contains a collection of various test suites. With additional exploration, we could infer the implementation for that feature is within `battery` under the directory of `blink/renderer/modules` that contains a collection of various renderer modules.

4.2.2 Feature-Oriented Call Graph Generation.

Function Identifier. Once the above initial mapping is complete, SLIMIUM constructs a call graph based on IR functions. Recall that we aim to directly remove binary functions on top of the mapping information; hence SLIMIUM instruments the final Chromium binary by assigning a unique identifier for each IR function at its build. The discrepancy between IR and binary functions happens because of i) object files irrelevant to the final binary at compilation (i.e., assertions that ensure a correct compilation process or platform-specific code) and ii) de-duplication at link time [19] (i.e., a single constructor or destructor of a class instance would be selected if redundant) [7]. It is noted that we can safely discriminate all binary functions because they are a subset of IR functions.

Indirect Call Targets. As Chromium is mainly written in a C++ language, there are inherently a large number of indirect calls, including both virtual (91.8%) [45] and non-virtual calls. Briefly, we tackle identifying indirect call targets using the following two techniques: i) backward data analysis for virtual calls and ii) type matching for non-virtual calls. In case of failing the target in a backward data analysis for a virtual call invocation, we attempt to use type matching. Note that it is infeasible to obtain all indirect call targets with full accuracy. We describe the whole call graph construction in Section 5.

4.2.3 Feature-Code Mapping with Relation Vectors.

At this point, we have a large directed call graph (nodes labeled with a function identifier and edges that represent caller-callee relationships) and an initial mapping between a feature and corresponding source files. Even with the mapping information alone, it is possible to learn partial binary functions that belong to relevant object files; however, there are quite a few missing links, including binary functions from sources generated at compilation (Section 2.4).

To seek more relevant object files for each feature, we define a two-dimensional *relation vector*, $\vec{R} = (r_c, r_s)$, which represents the following two vector components: i) call invocations (r_c) and ii) similarity between two object file names using the hamming distance [49] (r_s). The relation vector serves as a metric on how intensively any two objects are germane to each other. The intuition behind this is that i) it is reasonable to include an object as part of a feature if function calls would be frequently invoked each other, ii) relevant code is likely to be implemented under a similar path name, and iii) a non-deterministic code path problem (i.e., exceptions) can be minimized by including all functions within an object.

Figure 4 illustrates three phases that automatically infer relevant code at the object level. First, as in Step I, we group a set of binary functions that belong to the same object (i.e., `f7` and `f8` with a dotted-line area). In this example, there are five objects grouped with nine functions total. Second, we build another directed graph for object dependencies (Step II) based on the edges from the previous function call graph. Each edge defines an *object-object relation vector*, $\vec{R_O} = (r_c, r_s)$, between two objects (nodes). For $\vec{R_O}$, each component can be computed as the number of call invocations and a hamming distance value. For instance, the R_O between `01` and `02` can be represented as $(2, 0.5)$ because of two function invocations (i.e., (`f2`) → (`f4`, `f5`)) and a hamming distance value of 0.5 from the two object names (i.e., `aabb` and `bbbb`). Third, we consider a feature on top of the object dependency graph (Step III). The initial mapping from manual discovery comes into play, which identifies relevant objects for a certain feature. Suppose the two objects, `02` and `03`, belong to Feature X (dotted-line in blue). Now, we compute another relation vector, a *feature-object relation vector*, $\vec{R_F} = (r_c, r_s)$ for the edges only connected to the feature. For $\vec{R_F}$, the r_c component represents the rate of call invocations between the feature and the surrounding objects (that have edges) whereas the r_s component is an amortized hamming distance value between the object name and the object name(s) that belong to the feature. In this example, the R_F between Feature X and the object `01` would be $(0.75, 0.25)$ because $r_c = \frac{2+1}{1+2+1}$ and $r_s = \frac{0.5+0}{2}$, respectively. Hence, the result can be interpreted as follows: the `01` has a high outgoing call invocation rate to the functions in Feature X; however, its object name is not close enough. Algorithm 1 briefly shows pseudo-code on how to explore any relevant objects for further debloating using relation vectors.

Note that we open both r_c and r_s as hyperparameters of our heuristic algorithm to determine the proximity between a feature and an object, ranging from 0 to 1. In our experiment, we use the value of 0.7 for both parameters (See Section 6.3 in detail).

4.3 Prompt Webpage Profiling

We employ a dynamic profiling technique to complement static analysis (i.e, feature-code mapping) because the granularity of our debloating approach aims at the function level. Various tools are available such as Dynamorio [6] and Intel Pin [24] to obtain exercised functions at runtime through dynamic instrumentation. However, instrumented code inevitably introduces considerable performance degradation that leads to a huge lag when running a giant application such as Chromium. Although a hardware-assisted means such as Intel PT [27] significantly addresses the performance issue during a trace, decoding the trace result is non-negligible (i.e., a couple of hours when visiting a certain webpage for a few seconds in Chromium).

Due to the impracticality of prior approaches, we devise a new means to trace with an in-house instrumentation, recording exercised functions akin to an AFL's [2] approach of a global coverage map. First, we allocate a shared memory that can cover the entire IR functions, a superset of binary functions. Second, we build Chromium so that it could mark every exercised function in the shared memory. Third, we *promptly* obtain the list of exercised functions by parsing the whole bits in the shared region after visiting a target webpage. We have not experienced any slowdown with the instrumented version of Chromium during our profiling

[7] In our experiment, almost half of IR functions were disappeared.

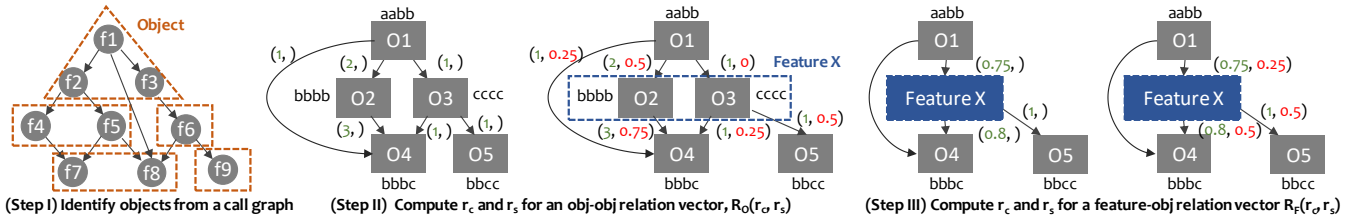

Figure 4: Feature-code mapping with relation vectors that enable the inference of relevant object files for a certain feature.

because our instrumentation requires merely a few bit-operations, a memory read, and a memory write for each function.

As discussed in Section 3.1, it is highly likely to trigger divergent execution paths due to Chromium's inherent complexity even when loading the same page again. We tackle this problem simply by reloading a webpage multiple times until reaching a point when no more new exercised functions are observed with a fixed sliding window (i.e., the length of revisiting that does not introduce a new exercised function; 10 in our case). With the Top 1000 Alexa websites [3], we had to visit a main page of each site approximately 172 times on average. Note that we leave this sliding window open as a hyperparameter.

5 IMPLEMENTATION

Our prototype SLIMIUM has been implemented with 111 lines of C, 1,140 lines of C++, and 1,985 lines of python code. We use the Chromium version of "77.0.3864.0" for our experiment. The key enablers of our debloating are the three LLVM passes [17] as follows.

Call Graph Construction. We develop the first LLVM pass that performs static analysis on the bitcode to construct an entire call graph in Chromium. It is straightforward to resolve direct calls by parsing the `CallInst`/`InvokeInst` in LLVM. However, it is challenging to identify indirect calls such as virtual calls.

Simply put, a virtual call requires obtaining both a virtual table that stores function pointers for a class and an index (i.e., offset) of the table. First, we build a class hierarchy graph where a node and edge represent a class and inheritance relationship between classes, respectively. Second, we perform intra-procedural backward data analysis on IR instructions, attempting to acquire a virtual table and an offset for each virtual call. With the class hierarchy graph, we explore child classes and have them share virtual tables with their parents. It is possible to extract a target function for the virtual call when the offset is available. Otherwise, we leverage type matching (i.e., parameter and return types) to identify the targets in case the offset is unknown for other reasons (i.e., the offset determined at runtime or it does not reside in the current function stack). Likewise, we take advantage of the type matching to find targets for non-virtual calls as well. In the end, our LLVM pass obtains an approximate call graph for all binary functions.

Chromium Instrumentation for Profiling. We develop the second LLVM pass that enables us to record exercised functions. We create a global variable, initialized to point to a shared memory, which contains a bitmap for marking the exercised functions. In our experiment, we need to pre-allocate around 1MB for 951K IR functions. Our pass inserts the instructions at the beginning of each function, which i) fetches the shared memory pointer, ii) moves it forward to a bit that represents the current function, and iii) sets the bit to one. To avoid multiple reads and writes for the shared memory whenever a function is exercised, the pass also allocates another global variable for each function that indicates whether the function has been recorded. The inserted instructions are responsible for setting each bit (initially set to 0) for the function's global variable to one when each function has been exercised (a single time check). Finally, we need to statically instrument Chromium for recording exercised functions with the above LLVM pass.

Function Boundary Identification. We develop the third LLVM pass to build Chromium such that it is capable of identifying function boundaries during binary rewriting. The pass inserts a store instruction to assign a unique identifier (i.e., an integer) for each IR function. It is significant that we orchestrate the pass to run after every optimization has been completed on LLVM IR level because our pass should not interfere with other optimization techniques. Note that the final Chromium binary might slightly differ from the one without our pass because an LLVM optimization scheduler may choose a different set of optimizations during the compilation. For example, some empty functions have not been eliminated because of the inserted instructions. We collect binary functions that have been compiled into the final Chromium binary from disassembling the binary and extracting the inserted instructions rather than relying on debugging information.

Binary Instrumentation. As a final step, once the feature-code mapping and profiling results are ready, SLIMIUM generates a debloated Chromium mutation after removing unnecessary features. Note that SLIMIUM maintains not only all other code regions except feature-relevant code but also non-deterministically exercised code at the function level even when a feature turns out not to be in use (with our threshold). Code elimination is trivial because we nullify unused code with illegal instructions based on known binary function boundaries. Once the instructions triggers a Chromium's error handling routine that catches an exception, an error page shows an "Aw, Snap!" message by default instead of crashing a whole Chromium process. SLIMIUM can produce a Chromium variant within *less than a second*. Currently, we only support an ELF (Executable and Linkable Format) binary format. We develop our own binary rewriter in pure Python without any dependency on the third party library.

6 EVALUATION

In this section, we evaluate SLIMIUM on a 64-bit Ubuntu 16.04 system equipped with Intel(R) Xeon(R) E5-2658 v3 CPU (with 48 2.20 GHz

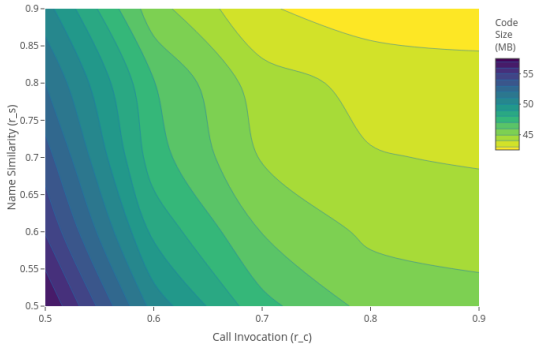

Figure 5: Contour plot of additionally discovered code size with a set of different relation vectors $\vec{R} = (r_c, r_s)$.

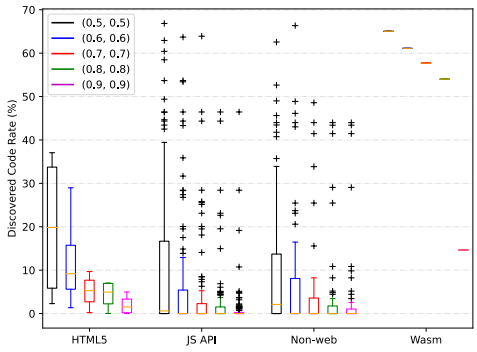

Figure 6: Breakdown of additional code discovery rates for each feature group across different relation vectors.

cores) and 128 GB RAM. In particular, we assess SLIMIUM from the following three perspectives:

- Correctness of our discovery approaches: How well does a relation vector technique discover relevant code for feature-code mapping (Section 6.1) and how well does a prompt web profiling unveil non-deterministic paths (Section 6.2)?
- Hyperparameter exploration: What would be the best hyper-parameters (thresholds) to maximize code reduction while preserving all needed features reliably (Section 6.3)?
- Reliability and practicality: Can a debloated variant work well for popular websites in practice (Section 6.4)? In particular, we have quantified the amount of code that can be removed (Section 6.4.1) from feature exploration (Section 6.4.2). We then highlight security benefits along with the number of CVEs discarded accordingly (Section 6.4.3).

6.1 Code Discovery with a Relation Vector

Our investigation for the initial mapping between features and source code requires approximately 40 hours of manual efforts for a browser expert. We identify 164 features and 6,527 source code files that account for 41.1 MB (37.4%) of the entire Chromium binary. Then, we apply a relation vector technique to seek more objects, as described in 4.2.2. To exemplify, the initial code path for `Wasm` only indicates the directory of `v8/src/wasm`, however, our technique successfully identifies relevant code in `v8/src/compiler`. In this case, although the object names under that directory differ from the beginning directory (i.e., `wasm`), the call invocation component of the vector correctly deduces the relationship.

Figure 5 concisely depicts that varying threshold pairs of name similarity (r_s) and call invocation (r_c) are inversely proportional to additional code discovery at large. The dark blue area on the lower left corner holds relatively a high value (i.e., 57.0 MB for $(0.5, 0.5)$) whereas the yellow area on the upper right holds a low one (i.e., 42.3 MB for $(0.9, 0.9)$). Similarly, Figure 6 shows a distribution of additional code discovery rate with a handful of different threshold sets from $(0.5, 0.5)$ to $(0.9, 0.9)$. The boxplot implies a moderate variance with outliers, but the medians consistently decrease at all four groups when raising those parameters because it means less code would be included for a feature, intolerant of fewer call invocations and dissimilar path names. For non-web features, the median is close to zero because of many features from third-party libraries, which implies that those components have a minimal dependency. In our experiment, we set both hyperparameters (r_c and r_s) to 0.7, resulting in a 9.3% code increase (41.1 → 44.9 MB) on average, each of which breaks down into 4.9%, 8.6%, 5.0%, and 57.7% for HTML5, JSAPI, Non-Web, and Wasm, respectively. We discuss how to select the best hyperparameters for SLIMIUM in Section 6.3.

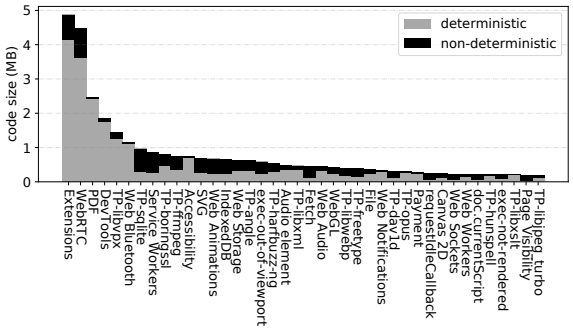

Figure 7: Ratio between non-deterministic code (dark bars on top) and the rest for the selected features. A prefix of `TP_` represents a third-party component.

6.2 Non-deterministic Paths Discovery with Webpage Profiling

To identify non-deterministic code paths, we performed webpage profiling for the Top 1000 Alexa websites in an automated fashion: opening a main page of each site in Chromium, waiting for 5 seconds to load, exiting, and repeating until no more exercised functions are found via a differential analysis. Our empirical results show that it requires continuous visits of 172 times on average.

Figure 7 illustrates non-deterministic portions of whole code from the top 40 debloatable features in size. The rate varies depending on each feature. On the one hand, network (`Service Workers`, `Fetch`), local cache (`third_party_sqlite`, `IndexedDB`) and animation features (`SVG`, `Web Animations`) trigger a substantial portion of non-deterministic code, mostly because they are designed for

flexible behaviors (i.e., networking, caching, animation). For example, `Service Workers` acts as proxy servers, which heavily relies on network status and caching mechanism, and `Fetch` fetches resources across the network. In a similar vein, `third_party_sqlite` and `IndexedDB` store data on a local machine for future services. Interestingly, we observe that non-deterministic code rates of `SVG` and `Web Animations` are also high because, in general, advertisements employ those features to trace any changes whenever a page is reloaded. On the other hand, features such as `PDF`, `Web Bluetooth`, and `Accessibility` maintain a low non-deterministic code rate (i.e., opening the same PDF document).

6.3 Hyperparameter Tuning

In this section, we explore four tunable hyperparameters (thresholds) for SLIMIUM: i) two relation vector components: call invocation (r_c) and name similarity (r_s) (Section 4.2.3), ii) code coverage rate (T), and iii) sliding window size, that is, the length of revisiting times that no new function has been exercised. We empirically set up the sliding window as 10 described in Section 4.3, hence here focuses on the other three thresholds.

In particular, T is an important factor that determines whether a feature can be a candidate to be further eliminated based on the portion of exercised code. This is because i) part of feature code may contain initialization or configuration for another and ii) a small fraction of feature code may be exercised for a certain functionality. Under such cases, we safely maintain the exercised functions of that feature instead of removing the whole feature code. To summarize, we keep entire code (at the feature granularity) if code coverage rate is larger than T or exercised code alone (at the function granularity) otherwise.

Figure 8 depicts the size of eliminated code on average by loading the front page of the Top 1000 Alexa websites to explore the best hyperparameters, namely $\vec{R_F} = (r_c, r_s)$ and T, empirically. Each subfigure represents the size of code reduction (y-axis) depending on a different combination of r_c (x-axis) and r_s (line) where both r_c and r_s range from 0.5 to 0.9, and T is a fixed value (ranging from 10% to 45%). One insight is that a higher T value improves code reduction because unexercised code for more features has been removed; e.g., 11.9 MB code removal with (r_c, r_s, T) = (0.7, 0.7, 0.1) whereas 27.3 MB with (0.7, 0.7, 0.45). Another insight is that a lower pair of (r_c, r_s) often improves code reduction. For example, 30.1 MB code has been removed where T is 30% with (r_c, r_s) = (0.5, 0.5) while only 24.9 MB with (r_c, r_s) = (0.7, 0.7). However, sometimes a lower pair of (r_c, r_s) does not improve code reduction because of the dynamics of our code discovery process; e.g., the first three subfigures indicate that code reduction with 0.6 of r_c is not smaller than the one with 0.5 or 0.55.

However, higher code reduction may decrease reliability of a debloated variant because it increases the chance of erasing non-deterministic code. As it is significant to strike a balance between code elimination and reliable binary instrumentation, we finally choose (r_c, r_s, T) = (0.7, 0.7, 0.3) with the following three observations. First, there is little impact on code reduction increase when T reaches up to around 25%. Second, with a r_s fixed, code reduction decreases from $r_c = 0.7$ heavily (i.e., T is 25%, 30% or 35%). Third, with a r_c fixed, code reduction slightly drops from $r_s = 0.7$. Exploration for a different combination of hyperparameters per feature looks also promising, which is open as part of our future research.

6.4 Chromium Debloating in Practice

In this section, we choose 40 popular websites from 10 categories to thoroughly assess reliability and security benefits of our debloating framework in practice instead of just loading the main page of a website. Table 2 summarizes a series of user activities of each website and experimental results for both code and CVE reduction.

6.4.1 *Code Reduction and Reliability.*

Code Reduction. Table 2 shows empirical code reduction with a debloated Chromium that allows a limited number of websites per each category. It removes 53.1% (23.85 MB) of the whole feature code on average with a single exception of `Remote Working` category (41.4% removal) because it harnesses `WebRTC` (See Section 6.4.2 in detail). Likewise, a debloated mutation that supports all 40 websites removes 38.8% code (around 17.4MB) as the last line of Table 2.

Next, we evaluate code reduction relevant to security features including four major ones that are fundamentally important to a modern web environment: same origin policy (SOP), content security policy (CSP), subresource integrity (SRI) and cross-origin resource sharing (CORS). Based on our manual profiling results, all 40 websites employ these four features where code coverage on average are 8.3%, 39.1%, 34.0% and 79.0% for SOP, CSP, SRI and CORS, respectively. Since SOP, CSP and SRI are not part of our feature set, SLIMIUM offers corresponding security features at all times. Although CORS is part of our feature-code map, we observe heavy use of this feature for the websites in our experiment (i.e., min/max code coverages are 60.5%/85.8%), thus SLIMIUM does not remove any code. We have other security related features in our feature-code map, such as `Credential Management`, `FIDO U2F`, and `Web Cryptography`. The corresponding code may be possibly removed because of low code coverage; for example, none of the 40 websites uses the feature of `FIDO U2F` based on our profiling results. In this case, if a website would trigger any code for a security feature removed by SLIMIUM, a debloated Chromium variant would throw an illegal instruction exception and stop loading a page rather than allow one to visit a website without that security feature.

Reliability. We have repeated the same activities (from initial webpage profiling) using different mutations, resulting in *flawless browsing for all cases without any crash.* As a case study, we investigate three websites in a `Remote Working` category that offer their services with native applications on top of a Chromium engine. Both `Slack` and `Bluejeans` are built with an Electron framework [10] (embedded Chromium and *Node.js*), containing 109.4 MB and 111.6 MB code, respectively, where `Zoom` contains 99.6 MB. Compared to those applications, our debloated version for `Remote Working` sorely contains 91.4 MB (up to 18.1% code reduction) that maintains every needed functionality. Note that `Webex` has been ruled out because it runs on Java Virtual Machine.

With the different debloated versions of Chromium, we were able to visit all 40 websites flawlessly thanks to non-deterministic code identification and appropriate hyperparameter selection ((r_c, r_s, T) = (0.7, 0.7, 0.3)). However, theoretically it is possible to encounter

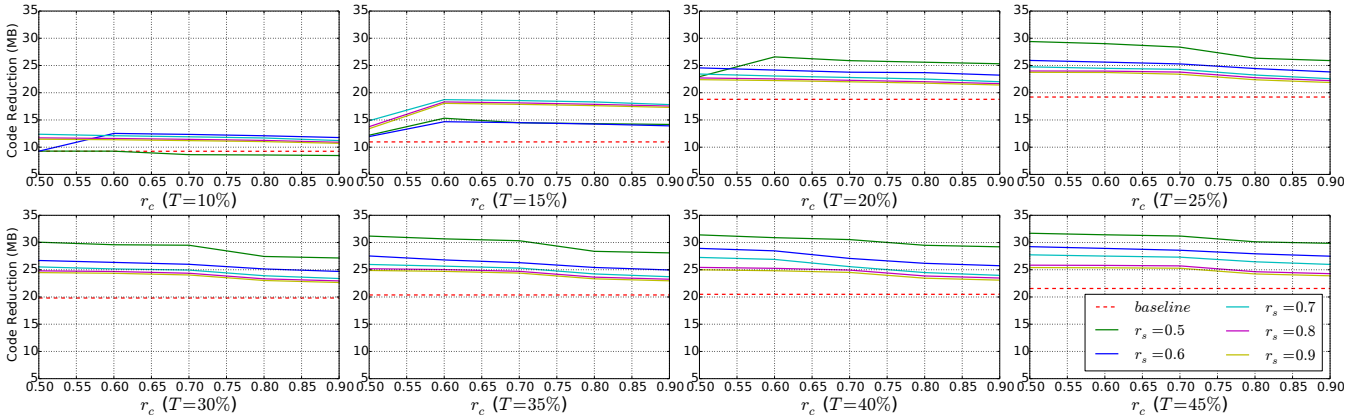

Figure 8: Average code reduction with a combination of different thresholds (r_c: call invocation, r_s:name similarity, T: code coverage rate) when loading the front page of the Top 1000 Alexa websites. The `baseline` represents the size of code reduction based on the initial feature-code map before applying Algorithm 1 in Appendix.

Table 2: Code and CVE reduction across debloated variants of Chromium per each category (See Figure 9 in detail).

Category	Websites	User Activities	Code Reduction Size (MB)	Code Reduction Rate (%)	Number of Removed CVEs
Airline	aa, delta, spirit, united	Login; search a flight; make a payment; cancel the flight, logout.	24.17	53.8	97
Email	gmail, icloud, outlook, yahoo	Login; read/delete/reply/send emails (with attachments); open attachments; logout.	23.75	52.9	97
Financial	americanexpress, chase, discover, paypal	Login; check a statement; pay a bill; transfer money; logout.	23.45	52.2	91
News	cnn, cnbc, nytimes, washingtonpost	Read breaking news; watch videos; search other news.	24.19	53.9	98
Remote Working	bluejeans, slack, webex, zoom	Schedule a meeting; video/audio chat; share a screen; end the meeting.	18.57	41.4	81
Shopping	amazon, costco, ebay, walmart	Login; track a previous order; look for a product; add it to the cart; checkout; logout.	24.33	54.2	98
Social Media	instagram, facebook, twitter, whatsapp	Login; follow/unfollow a person; write a post and comment; like a post; send a message; logout.	23.30	51.9	93
Sports	bleacherreport, espn, nfl, nba	Check news, schedules, stats, and players.	24.39	54.3	98
Travel	booking, expedia, priceline, tripadvisor	Login; search hotels; reserve a room; make a payment; logout.	24.16	53.8	97
Video	amazon, disneyplus, netflix, youtube	Search a keyword; play a video (forward/pause/resume); switch screen modes (normal/theatre/full) ; adjust a volume	24.18	53.9	93
All	–	–	17.43	38.8%	73

a false positive with a hyperparameter set of other choice. For example, by increasing T from 0.3 to 0.35, a debloated Chromium mutation would fail to load `Bluejeans` and `CNN` with the accidental removal of `Web Audio API`. Similarly, we observe additional failures of `Nytimes`, `Washingtonpost`, and `Whatsapp`, with the removal of `Web Workers` when T=0.4.

6.4.2 ***Feature Exploration.*** Figure 9 depicts the heatmap for actual code coverage rates across different features per each category in Table 2 at a glance. Note that common features have been eliminated to show a distinct feature usage alone. The websites from the `Remote Working` group clearly adopt several unique Web features designed for Real-Time Communication (RTC) that are hardly seen from the others, including `WebRTC` and `Media Stream`, and the third party libraries (i.e., `webrtc_overrides`, `usrsctp`, `opus`, `libsrtp`). We also confirm a handful of interesting instances based on our activities as follows. A `PDF` feature has been rarely used but `Financial` because of opening PDF documents (i.e., bank statements). `Financial` and `Travel` have harnessed `Accelerometer`, `Gyroscope`, and `Orientation Sensor` for checking device orientation. `Remote Working` and `Video` sorely adopt `Full Screen` due to switching to a screen mode. Most websites employ the `libphonenumber` feature to maintain personal information with a login process whereas `News` and `Sports` do not. All the above examples explain that inner components in Chromium have been well-identified for the debloating purpose.

6.4.3 ***Security Benefits.*** To confirm the security benefits of our approach, we have collected 456 CVEs pertaining to Chromium from online resources [9, 32] and official (monthly) security updates from Google. We focus on the rest of the 364 CVEs that have been patched for the last two years (Some of them might be assigned in previous years), excluding 92 of them in case of i) no permission to access CVE information or ii) a vulnerability relevant to other platforms (i.e, iOS, Chrome OS). Although it may be less useful to count CVEs for a single version of Chromium because different versions are generally exposed to a different CVE set, we include them to evaluate the effectiveness of debloated mutations. It is noteworthy

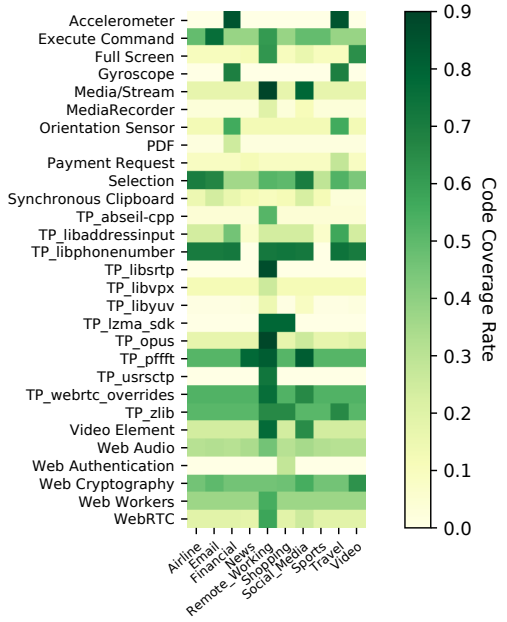

Figure 9: Code coverage rate of various features across different websites. A prefix of `TP_` represents a third-party component.

that we check out both vulnerable code and corresponding patches for mapping the CVEs to affected features.

Table 3 in Appendix summarizes Chromium CVEs by 13 different vulnerability types and three severity levels that are associated with features for debloating. We adopt the severity level (i.e., High, Medium, Low) of each CVE calculated by the National Vulnerability Database (NVD) [32]. The most common vulnerability type is use-after-free (52), followed by overflow (46), and insufficient policy enforcement (43). Other vulnerability types include uninitialized memory, XSS (cross site scripting), and null dereference. Interested readers may refer to the full list of Chromium CVEs in Appendix Table 5. Note that 153 (out of 364) CVEs are associated with 42 features in our feature-code map. From our experiments in Table 2, around 94 out of 153 potential CVEs (61.4%) have been removed on average when visiting the websites of our choice at each category. The number of CVEs that has been eliminated for the group of `Remote Working` is relatively low predominately due to RTC features. Slimium successfully removes 73 CVEs (47.7%) across all 40 websites.

7 DISCUSSION

In this section, we discuss the limitations and applications.

7.1 Limitations

We have shown the effectiveness and robustness of Slimium in removing unnecessary code for Chromium with a feature subsetting technique. However, Slimium relies on a hybrid analysis including static, dynamic, and various heuristic means. As none of the above suffices for either completeness or soundness (i.e., incomplete call graph, a heuristic mean to discover feature code, hyperparameter setting for profiling), we have limitations from the following aspects: i) dynamic nature of the web itself and ii) a failure of the above analysis.

From the first aspect, Slimium might be less useful if the content on the server side is dramatically altered (e.g., website reconstructions or updates that may employ completely different features). Our approach assumes that features at the time of profiling are not much different from the ones at the time of instrumentation. One possible solution may be automatic profiling of target websites on a regular basis to detect such considerable changes. Another concern is that our technique becomes ineffective if a website already has been compromised to exploit a certain feature at the first visit. From the second aspect, Slimium is capable of taking both conservative and aggressive approaches on purpose by leaving tunable hyperparameters such as relation vector components, sliding window for non-deterministic path discovery, and feature code coverage rate. Although Slimium generates debloated versions of Chromium that work flawlessly in our experiment (Section 6.4.1), it is possible that Slimium would mistakenly eliminate needed code. We believe machine learning techniques could assist probabilistic decision making such as thresholds or feature code exploration, which is part of our future research. Besides, supporting features may vary depending on versions of Chromium and Slimium relies on manual analysis during feature-code map generation in the beginning. Therefore, keeping an eye on updates is another key to experience the best debloating result. Finally, it is desirable to develop an automatic process for building a feature-code map, applying Slimium to different versions of Chromium with a tremendous difference and other large software with modularized code structures.

One may raise a concern that blind code removal arises a new vulnerability because any code relevant to security checks could be eliminated in case that profiling does not capture them (e.g., bound checks to prevent memory corruption, access control checks). However, since Slimium rewrites unused code with illegal instructions, a debloated version of Chromium would not introduce a new vulnerability from such missing security checks but trigger exceptions instead. We believe that machine learning and pattern matching can identify code relevant to security checks, which we leave as part of our future work.

7.2 Applications

The conceivable application of Slimium is twofold: from an end user and developer perspective. One of promising applications would be a restricted browsing in a public place (e.g., library, hotel, kiosk) or government departments (e.g., DoD, CDC), which requires an access to a set of authorized websites only. Although there are other means to limit the access, Slimium can provide a systematic way for secure browsing that cannot be circumvented through direct elimination of unneeded functionality. Another fruitful case with Slimium would be supporting a development framework based on a Chromium engine such as Electron [10] that produces environment-agnostic applications. It helps a developer to generate the slim version of an application with a subset of features by integrating Slimium with such frameworks. We anticipate that a profiling process would be more straightforward because the developer is aware of the needed features for an application much better.

8 RELATED WORK

It is crucial to identify accurately unneeded code to avoid accidental removal. To that end, prior works largely fall into three categories to identify unused features: a) binary analysis (*e.g.*, static or dynamic analysis), b) supplementary information with the help of a compilation toolchain, and c) machine learning techniques.

Debloating with binary analysis. One of the early works based on static analysis is CodeFreeze [31]. It presents a technique, dubbed "code stripping and image freezing" that eliminates imported functions not in use at load time, followed by freezing the remaining code to protect it from further corruption (*e.g.*, injecting code). Because it targets executable binaries whose sources are unavailable, this approach performs code removal atop a conservative static analysis. DamGate [53] introduces a framework to customize features at runtime. It leverages a handful of existing tools to build a call graph through both static and dynamic analyses. In a similar vein, TRIMMER [15] begins with identifying unnecessary features based on an user-defined configuration, followed by eliminating corresponding code from interprocedural analysis statically.

Meanwhile, Shredder [28] aims to filter out potentially dangerous arguments of well-known APIs (*e.g.*, assembly functions). It first collects the range of API parameters that a benign application takes and then enforces a policy to obey the allowable scope of the parameters from initial analysis. For example, a program would be suspended upon a call invocation with unseen arguments. Both FACE-CHANGE [55] and Kernel tailoring [22, 23] apply the concept of debloating to the kernel. The former makes each application view its own shrinking version of the kernel, facilitating dynamic switching at runtime across different process contexts, whereas the latter automatically generates a specific kernel configuration used for compiling the tailored kernel. Razor [38] proposes a heuristic technique to infer additional code paths that contain expected features based on exercised code. However, this work cannot directly be applied to Chromium debloating due to a large performance overhead while analyzing the execution trace after its collection with `Intel PT`, and the limitations of the heuristics used for inferring additional code paths. Meanwhile, both the bloat-aware design [7] and JRed [52] apply program customization techniques to Java-based applications; whereas Babak et al. [4] propose a debloating technique for web applications.

Debloating with supplementary information. Another direction toward code debloating takes advantage of a compilation toolchain to obtain additional information. Piece-Wise [40] introduces a specialized compiler that assists in emitting call dependencies and function boundaries within the final binary as supplementary information. The modified loader then takes two phases (i.e., page level and function level) to invalidate unneeded code at load time.

Debloating with machine learning techniques. Recent advancements in debloating leverage various machine learning techniques to identify unused code or features. CHISEL [16] produces a transformed version that removes unneeded features with reinforcement learning. Because it relies on test cases as an input to explore internal states, it might suffer from incorrect results when running a mutation that encounters an unexpected state. Hecate [51] leverages deep learning techniques to identify features and corresponding functions. It uses both a recursive neural network (RNN) to compute semantic representation (*e.g.*, unique embedding vector per each opcode and operand) and a convolutional neural network (CNN) for a function mapping as a multi-class classifier. Binary control flow trimming [11] introduces a contextual control flow graph (CCFG) that enables the expression of explicitly user-unwanted features among implicitly developer-intended ones, learning a CCFG policy based on runtime traces. BlankIt [34] applies machine learning to predicate functions needed at load time.

Other efforts to reduce attack surface. The work of Snyder et al. [42] is probably the closest in spirit to our work in that they leverage Web API standards to limit the functionality of a website. However, it has two major differences: a) the attack surface only contains standard Web APIs without considering non-web features, and b) the hardening mechanism lies in disabling specific features by intercepting JavaScript, implemented as one of the browser extensions. The actual implementation code still resides in memory; thus it could be circumvented [43] with an expected access. On the contrary, our browser hardening nullifies actual binary functions corresponding to a unit feature for debloating.

Anh et al. [39] propose bloat metrics for the first time to systematically quantify the bloatness of each program, library, and system library. CARVE [5] takes an approach of debloating unused source code, which requires both open source and a rebuilding process. Cimplifier [41] demonstrates that a container image could shrink its size up to 95%, preserving its original functionalities. Recently, Microsoft released `ApplicationInspector` [26], an attack surface analysis tool based on known patterns, automatically identifying third-party software components that might impact security. Other efforts include code removal based on configurable features for applications [21], system call specialization [13], and its policy generation [12] for containers.

9 CONCLUSION

This paper presents a novel debloating framework, Slimium, for a browser that leverages a hybrid approach for generating a slim version that removes unused features. We target Chromium, the most popular and leading browser that supports numerous features and thereby unavoidably exposes a wide range of attack vectors from its large code base. We introduce feature subsetting whose main idea is to determine a set of predefined features as a debloating unit. To this end, we devise a relation vector technique to build a feature-code map and a new means to enable a prompt webpage profiling to tackle the limitations of classic approaches. Our experimental results demonstrate the practicality and feasibility of Slimium, which eliminates 94 CVEs (61.4%) by cutting off 23.85 MB code (53.1%) of the whole feature code.

10 ACKNOWLEDGMENT

We thank the anonymous reviewers, and our shepherd, Nick Nikiforakis, for their helpful feedback. This research was supported by the ONR under grants N00014-17-1-2895, N00014-15-1-2162 and N00014-18-1-2662. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ONR.

REFERENCES

[1] Feross Aboukhadijeh. 2012. Using the HTML5 Fullscreen API for Phishing Attacks. https://feross.org/html5-fullscreen-api-attack/.

[2] AFL 2020. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: 2020-2-12.

[3] Alexa. 2020. The top 500 sites on the web. https://www.alexa.com/topsites.

[4] 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)*.

[5] Michael D. Brown and Santosh Pande. 2019. CARVE: Practical Security-Focused Software Debloating Using Simple Feature Set Mappings. In *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*.

[6] Derek Bruening and Saman Amarasinghe. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science.

[7] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J. Carey. 2013. A bloat-aware design for big data applications. In *Proceedings of the 2013 international symposium on memory management (ISMM)*.

[8] Caniuse.com. 2020. Support tables for HTML5, CSS3, etc. https://caniuse.com/#feat=feature-policy.

[9] CVE Details. 2019. Vulnerabilities statistics on Google Chrome. https://www.cvedetails.com/product/15031/Google-Chrome.html?vendor_id=1224.

[10] Electron. 2020. . https://www.electronjs.org/.

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

[12] 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)*.

[13] 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)*.

[14] Google. 2018. Introduction to Feature Policy. https://developers.google.com/web/updates/2018/06/feature-policy#list.

[15] Ashish Gehani Hashim Sharif, Muhammad Abubakar and Fareed Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*.

[16] 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)*.

[17] LLVM Compiler Infrastructure. [n.d.]. Writing an LLVM Pass. http://llvm.org/docs/WritingAnLLVMPass.html.

[18] Zero Day Initiative. [n.d.]. Published Advisories. https://www.zerodayinitiative.com/advisories/published/.

[19] S. Kell, D. P. Mulligan, and P. Sewell. 2016. The missing link: Explaining ELF static linking, semantically. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[20] Steve Kobes. 2020. Life of a pixel. https://bit.ly/lifeofapixel.

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

[22] 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*.

[23] 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)*.

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[25] Net Marketshare. 2015. Browser Market Share. https://netmarketshare.com/browser-market-share.aspx.

[26] Microsoft. 2020. Application Inspector. https://github.com/microsoft/ApplicationInspector.

[27] misc-pt-site 2020. Intel Processor Trace Tools. https://software.intel.com/en-us/node/721535. Accessed: 2020-2-12.

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

[29] Mozilla. 2019. Content Security Policy (CSP). https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP.

[30] Mozilla. 2020. Feature-Policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Feature-Policy#Directives.

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

[32] National Institute of Standards and Technology. 2020. National Vulnerability Database. https://nvd.nist.gov/.

[33] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P. Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis. 2019. Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[34] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt Library Debloating: Getting What You Want Instead of Cutting What You Don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[35] The Chromium Projects. 2020. Getting Around the Chromium Source Code Directory Structure. https://www.chromium.org/developers/how-tos/getting-around-the-chrome-source-code.

[36] The Chromium Projects. 2020. User Experience. https://www.chromium.org/user-experience.

[37] The Chromium Projects. 2020. Web IDL in Blink. https://www.chromium.org/blink/webidl.

[38] 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*.

[39] Anh Quach and Aravind Prakash. 2019. Bloat Factors and Binary Specialization. In *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*.

[40] 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.

[41] 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)*.

[42] 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)*.

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

[44] StatCounter. 2020. Browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share/desktop/worldwide.

[45] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*.

[46] W3C. 2019. Feature Policy. https://w3c.github.io/webappsec-feature-policy/.

[47] W3C. 2020. All standars and drafts. https://www.w3.org/TR/.

[48] we are social. 2019. DIGITAL 2019: GLOBAL INTERNET USE ACCELERATES. https://wearesocial.com/blog/2019/01/digital-2019-global-internet-use-accelerates.

[49] Wikipedia. 2020. Hamming distance. https://en.wikipedia.org/wiki/Hamming_distance.

[50] Wikipedia. 2020. Pwn2Own. https://en.wikipedia.org/wiki/Pwn2Own.

[51] 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)*.

[52] Dinghao Wu Yufei Jiang and Peng Liu. 2016. Jred: Program customization and bloatware mitigation based on static analysis. In *Proceedings of the 40th Annual Computer Software and Applications Conference (ACSAC)*.

[53] Tian Lan Yurong Chen 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)*.

[54] ZDNet. 2018. Microsoft's Edge to morph into a Chromium-based, cross-platform browser. https://zd.net/2OUytjP.

[55] Xiangyu Zhang Zhongshu Gu, Brendan Saltaformaggio 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)*.

APPENDIX

Algorithm 1: Explore a pertinent object to a feature

```
Result: added_objects
in_nodes = feature.in_nodes; ;                          // Obtain the incoming/outgoing nodes
out_nodes = feature.out_nodes;
r_c, r_s = 0.7; ;                                       // Initialize hyperparameters
for node : in_nodes do
    sum_c, sum_in_c, sum_in_s, in_num = 0;
    for edge : node.out_edges do
        (c, s) = edge.relation;
        sum_c += c; ;                                   // Sum up call invocations
        if edge.end_node in feature.nodes then
            sum_in_c += c;
            sum_in_s += s; ;                            // Sum up hamming distance values
            in_num += 1;
        end
    end
    if sum_in_c / sum_c > r_c || sum_in_s / in_num > r_s then
        added_objects.add(node);
    end
end
for node : out_nodes do
    sum_c, sum_in_c, sum_in_s, in_num = 0;
    for edge : node.in_edges do
        ...
        if edge.start_node in feature.nodes then
            ...
        end
    end
    ...
end
```

Table 3: Summary of Chromium CVEs and relevant unit features for debloating.

Vulnerability Type	High	Medium	Low	Total	Relevant Features
Bad cast	1	0	0	1	-
Bypass	3	22	2	27	document.{currentScript, domain}, Page Visibility, requestIdleCallback, Extensions, Service Workers, Video, and Web Audio
Disclosure	1	16	1	18	Extensions, Media Source, third_party_boringssl, Timing, Video, and Web Audio
Inappropriate implementation	0	14	1	15	DevTools, Extensions, and PDF
Incorrect security, handling, permissions	9	31	0	40	Extensions, DevTools, Navigator, Service Workers, URL, URL formatter, Web Assembly, WebRTC, and XMLHttpRequest
Insufficient policy enforcement	3	36	4	43	Canvas 2D, createImageBitmap, DevTools, Extensions, Payment, WebGL, Service Workers, Shared Web Workers, Page Visibility, requestIdleCallback, createImageBitmap, and document.{currentScript, and domain}
Insufficient validation	5	11	0	16	DevTools, IndexedDB, PDF, WebGL, and Web Assembly
Out of bound read	2	12	0	14	PDF, third_party_sqlite, and WebRTC
Out of bound write	3	5	0	8	PDF and Web Assembly
Overflow	14	32	0	46	Blob, Canvas 2D, Media Stream, PDF, Web Assembly, Web SQL, WebGL, WebGPU, WebRTC, and third_party_{angle, icu, and libxml}
Spoof	1	26	3	30	DevTools, Extensions, Full screen, Media Stream, and Web Bluetooth
Type Confusion	5	2	0	7	PDF, SVG, and WebRTC
Use after free	16	36	0	52	DevTools, Extensions, File, File System, IndexedDB, Media Capture, MediaRecorder, PDF, Payment, Web Assembly, Web Audio, Web MIDI, WebRTC, createImageBitmap, execution-while-out-of-viewport, and third_party_{libvpx, and libxml}
Others	8	34	5	47	Canvas 2D, createImageBitmap, DevTools, Directory selection, Extensions, Full screen, PDF, Service Workers, Web Assembly, Web Audio, WebRTC, and third_party_ffmpeg
Total	71	277	16	364	

Table 4: Chromium features as a unit of debloating. The columns V, P, C, and E represent the number of CVEs, Feature Policy support, Chromium support, and Experimental flag support respectively (Yes: ●, No: ✗, Partial: ◗).

Class	Feature Name	Func.	Size (B)	V	P	C	E
	Accessibility	3,018	774,862		✗	●	✗
HTML5	Canvas 2D	1,153	254,979	3	✗	●	✗
	SVG	3,884	710,004	1	✗	●	✗
	Video	518	98,402	2	✗	●	✗
	WebGL	1,789	445,175	5	✗	●	✗
	WebGPU	470	80,706	5	✗	●	✗
	AbortController & AbortSignal	33	5,187		✗	●	✗
JS API	Accelerometer	13	887		●	●	●
	Ambient Light Sensor	16	1,936		●	●	●
	Autoplay	65	13,763		●	●	✗
	Background Sync	739	145,961		✗	●	●
	Base64	12	2,356		✗	●	✗
	Console	122	49,646		✗	●	✗
	Battery Status	123	13,889		●	●	✗
	Beacon	13	2,071		✗	●	✗
	BigInt	95	34,269		✗	●	✗
	Blob	850	205,670	1	✗	●	✗
	Broadcast Channel	116	17,948		✗	●	●
	Channel messaging	106	31,310		✗	●	✗
	Constraint Validation	100	12,812		✗	●	✗
	createImageBitmap()	613	116,031	4	✗	●	✗
	Credential Management	224	57,680		✗	●	✗
	Cross-Origin Resource Sharing	196	68,668		✗	●	✗
	crypto.getRandomValues()	6	642		✗	●	✗
	CSS.supports()	3	617		✗	●	✗
	Custom Event	10	1,134		✗	●	✗
	Device Events	341	48,431		✗	◗	✗
	Directory selection	324	53,452	1	✗	●	✗
	Do Not Track	9	1,403		✗	●	✗
	Document Object Model Range	89	37,579		✗	●	✗
	document.currentScript	1,073	230,083	3	✗	●	✗
	document.domain	929	209,395	3	●	●	✗
	document.evaluate & XPath	335	92,237		✗	●	✗
	document.execCommand()	6	1,282		✗	●	✗
	DOM Element	568	143,336		✗	●	✗
	DOM Parsing and Serialization	457	111,963		✗	●	✗
	Encrypted Media Extensions	374	67,794		●	●	✗
	execution-while-not-rendered	1,029	217,103		●	●	✗
	execution-while-out-of-viewport	2,621	587,719	1	●	●	✗
	Feature Policy	152	53,400		✗	◗	●
	Fetch	2,418	509,398		✗	●	✗
	FIDO U2F	1,456	467,216		✗	●	✗
	File	1,833	390,459	3	✗	●	✗
	Filesystem & FileWriter	947	169,129		✗	●	✗
	Full Screen	196	42,364	5	●	●	●
	Gamepad	709	143,215		✗	●	✗
	Geolocation	235	41,809		●	●	✗
	Gyroscope	13	919		●	●	●
	High Resolution Time	464	102,944		✗	●	✗
	IndexedDB	3,053	861,111	2	✗	●	✗
	Internationalization	73	45,243	1	✗	●	✗
	IntersectionObserver	191	44,669		✗	●	✗
	Intl.PluralRules	13	7,239		✗	●	✗
	Magnetometer	13	919		●	●	●
	Media Capture	198	52,530	1	●	●	✗
	Media Recorder	275	67,833	1	✗	●	✗
	Media Source Extensions	374	67,794	1	✗	●	✗
	Media Stream	83	14,057	2	✗	●	✗
	Mutation Observer	179	34,649		✗	●	✗
	Native Filesystem	433	104,131		✗	●	●
	navigator.hardwareConcurrency	1	19	1	✗	●	✗
	Network Information	64	8,960		✗	●	✗
	Orientation Sensor	54	7,314		✗	●	●
	oversized-images	59	18,273		●	●	✗
	Page Visibility	946	214,054	3	✗	●	✗
	Payment	841	232,875	3	●	◗	✗
	Permissions	494	119,978		✗	●	✗
	Picture-in-Picture	225	35,203		●	●	✗
	Pointer events	146	42,262		✗	●	✗
	Pointer Lock	14	1,786		✗	●	✗
	Push	629	134,239		✗	●	✗
	requestAnimationFrame()	85	19,503		✗	●	✗
	requestIdleCallback	1,175	260,757	3	✗	●	✗
	Resize Observer	127	28,509		✗	●	✗
	Screen Orientation	170	26,174		✗	●	✗
	Selection	511	156,461		✗	●	✗
	Server-sent events	75	11,329		✗	●	✗
	Service Workers	4,518	1,049,970	4	✗	●	✗
	Shared Web Workers	419	79,337	1	✗	●	✗
	Speech Recognition	843	229,729		✗	◗	✗
	Speech Synthesis	216	35,928		✗	●	✗
	Streams	698	126,798		✗	◗	✗
	Synchronous Clipboard	500	108,156		✗	◗	✗
JS API	TextEncoder & TextDecoder	68	15,068		✗	●	✗
	Timing	634	142,366	1	✗	●	✗
	Touch events	117	61,599		✗	●	✗
	URL	82	19,462	1	✗	●	✗
	Vibration	104	14,600		●	●	✗
	Wake Lock	142	20,954		●	●	✗
	Web Animations	2,524	705,348		●	◗	✗
	Web Audio	2,485	587,711	5	✗	●	✗
	Web Authentication	28	1,972		●	●	●
	Web Bluetooth	4,896	1,216,400	1	✗	●	●
	Web Cryptography	664	148,808		✗	●	✗
	Web MIDI	339	56,585	1	●	●	✗
	Web Notifications	1,539	342,313		✗	●	✗
	Web Sockets	921	195,979		✗	●	✗
	Web SQL	923	185,985	1	✗	●	✗
	Web Storage	3,177	698,731		✗	●	✗
	Web Workers	1,549	246,759		✗	●	✗
	WebRTC	16,993	4,797,715	8	✗	●	●
	WebUSB	577	131,635		●	●	✗
	WebVR	255	55,213		●	●	●
	WebXR	859	177,361		●	●	●
	Window	325	73,359		✗	●	✗
	XMLHttpRequest	164	41,196	1	●	●	✗
	DevTools	10,273	2,889,315	15	✗	●	✗
Non-web	Extensions	25,073	5,735,571	29	✗	●	✗
	NACL & PNACL	671	187,277		✗	●	✗
	PDF	8,253	2,601,303	26	✗	●	✗
	third_party_abseil-cpp	30	3,274		✗	●	✗
	third_party_angle	1,503	636,685	4	✗	●	✗
	third_party_boringssl	2,142	824,106	1	✗	●	✗
	third_party_breakpad	109	49,047		✗	●	✗
	third_party_brotli	42	34,158		✗	●	✗
	third_party_cacheinvalidation	891	187,681		✗	●	✗
	third_party_ced	46	50,842		✗	●	✗
	third_party_cld_3	332	82,148		✗	●	✗
	third_party_crc32c	3	3,209		✗	●	✗
	third_party_dav1d	422	316,415		✗	●	✗
	third_party_ffmpeg	1,453	795,694	1	✗	●	✗
	third_party_flac	148	86,028		✗	●	✗
	third_party_fontconfig	394	126,702		✗	●	✗
	third_party_freetype	776	400,392		✗	●	✗
	third_party_harfbuzz-ng	993	554,211		✗	●	✗
	third_party_hunspell	264	221,096		✗	●	✗
	third_party_iccjpeg	2	1,302		✗	●	✗
	third_party_icu	5,782	1,793,490	1	✗	●	✗
	third_party_inspector_protocol	197	85,215		✗	●	✗
	third_party_jsoncpp	79	37,373		✗	●	✗
	third_party_leveldatabase	608	180,080		✗	●	✗
	third_party_libaddressinput	281	92,267		✗	●	✗
	third_party_libjingle_xmpp	399	87,405		✗	●	✗
	third_party_libjpeg_turbo	325	196,271		✗	●	✗
	third_party_libphonenumber	235	85,665		✗	●	✗
	third_party_libpng	231	99,653		✗	●	✗
	third_party_libsrtp	121	34,315		✗	●	✗
	third_party_libsync	1	163		✗	●	✗
	third_party_libvpx	1,696	1,456,960	2	✗	●	✗
	third_party_libwebm	111	40,614		✗	●	✗
	third_party_libwebp	672	399,824		✗	●	✗
	third_party_libxml	977	498,403	2	✗	●	✗
	third_party_libxslt	394	216,254		✗	●	✗
	third_party_libyuv	455	172,021		✗	●	✗
	third_party_lzma_sdk	6	1,458		✗	●	✗
	third_party_modp_b64	2	774		✗	●	✗
	third_party_openscreen	1	67		✗	●	✗
	third_party_opus	304	313,488		✗	●	✗
	third_party_ots	220	141,284		✗	●	✗
	third_party_perfetto	720	183,312		✗	●	✗
	third_party_pffft	35	23,113		✗	●	✗
	third_party_re2	320	168,288		✗	●	✗
	third_party_s2cellid	8	2,280		✗	●	✗
	third_party_sfntly	1,350	114,962		✗	●	✗
	third_party_smhasher	6	2,386		✗	●	✗
	third_party_snappy	20	4,492		✗	●	✗
	third_party_sqlite	896	1,016,192	1	✗	●	✗
	third_party_tcmalloc	167	37,048		✗	●	✗
	third_party_unrar	384	175,136		✗	●	✗
	third_party_usrsctp	397	310,183		✗	●	✗
	third_party_webrtc_overrides	44	3,876		✗	●	✗
	third_party_woff2	18	24,982		✗	●	✗
	third_party_zlib	177	92,515		✗	●	✗
	wasm	2,723	1,893,353		✗	●	✗

Table 5: Chromium CVEs associated with our feature set. The severity column ranges from low(▏), medium(▌) to high(■).

Features	Category	Sev.	CVE	Features	Category	Sev.	CVE
Blob	Overflow	▌	2017-15416	PDF	Overflow	▏	2018-6120
Canvas 2D	Insufficient policy	■	2019-5766			▌	2017-15408
		■	2019-5814			▌	2018-17461
	Others	▌	2019-5787			▌	2018-17469
	Overflow	■	2018-18338			▌	2019-5792
	UAF	▌	2019-5758			▌	2019-5795
DevTools	Inappropriate implementation	▌	2018-18344			▌	2019-5820
	Incorrect security	▏	2018-6112			▌	2019-5821
		▌	2018-6112		Type Confusion	▌	2018-6170
		▌	2018-6139		UAF	■	2017-15410
	Insufficient policy	▌	2017-15393			■	2017-5127
		▌	2019-5768			■	2018-18336
	Insufficient validation	■	2018-6101			■	2019-5762
		▌	2018-6039			▌	2017-15411
		▌	2018-6046			▌	2017-5126
	Others	▌	2018-6152			▌	2018-6088
	UAF	■	2018-6111			▌	2019-5756
		▌	2018-6111			▌	2019-5772
Directory selection	Others	▌	2018-6095			▌	2019-5805
document.*	Bypass	▌	2019-5799			▌	2019-5868
		▌	2019-5800	Service Workers	Bypass	▌	2018-6093
	Insufficient policy	▌	2018-18350		Incorrect security	▌	2018-6091
	UAF	■	2019-5759		Insufficient policy	▌	2019-5779
Extensions	Bypass	▌	2018-6070		Others	▌	2019-5823
		▌	2018-6089	Shared Web Workers	Insufficient policy	▌	2018-6032
	Disclosure	■	2018-6179	SVG	Type Confusion	▌	2019-5757
	Inappropriate implementation	■	2018-20065	third_party_angle	Overflow	■	2018-17466
	Incorrect security	■	2018-16064			▌	2019-5806
		■	2019-5793			▌	2019-5817
		▌	2017-15420			▌	2019-5836
		▌	2018-6110	third_party_boringssl	Disclosure	■	2017-15423
	Insufficient policy	■	2019-13754	third_party_ffmpeg	Others	■	2017-1000460
		▏	2018-6045	third_party_icu	Overflow	■	2017-15422
		▌	2017-15391	third_party_libvpx	UAF	▌	2018-6155
		▌	2017-15394			▌	2019-5764
		▌	2018-6035	third_party_libxml	Overflow	▌	2017-5130
		▌	2019-13755		UAF	▌	2017-15412
		▌	2019-5778	third_party_sqlite	OOB read	▌	2019-5827
	Others	■	2019-5838	Timing	Disclosure	▌	2018-6134
		▌	2018-16086	URL	Incorrect security	▌	2019-5839
		▌	2018-6121			▌	2018-6042
		▌	2018-6138	Video, Web Audio	Bypass	▌	2018-6168
		▌	2018-6176		Disclosure	▌	2018-6177
		▌	2019-5796	Web Assembly	Incorrect security	▌	2018-6116
	Spoof	▌	2019-13691			▌	2018-6131
	UAF	■	2018-20066		Insufficient validation	▌	2018-6036
		■	2018-6054		OOB write	▌	2017-15401
		▌	2018-20066		Others	■	2017-5132
		▌	2019-5878			▌	2018-6061
Extensions, DevTools	Incorrect security	▌	2018-6140		Overflow	▏	2018-6092
	Others	▌	2018-16081		UAF	▌	2017-15399
	Spoof	▌	2018-6178	Web Audio	Bypass	▌	2018-6161
File	UAF	▌	2018-6123		Others	■	2018-16067
		▌	2019-5786		UAF	■	2018-18339
		▌	2019-5788			■	2018-6060
File System		▌	2019-5872			▌	2017-5129
Full screen	Others	▌	2018-17471	Web Bluetooth	Spoof	■	2018-16079
		▌	2018-17476	Web MIDI	UAF	▌	2019-5789
	Spoof	▌	2017-15386	Web SQL	Overflow	■	2018-20346
		▌	2018-16080	WebGL	Insufficient policy	▌	2018-6047
		▌	2018-6096		Insufficient validation	■	2018-6034
IndexedDB	Insufficient validation	▌	2019-5773		Overflow	■	2017-5128
	UAF	▌	2019-13693			▌	2018-6038
Media Capture		■	2017-15395			▌	2018-6162
Media Source Extensions	Disclosure	■	2018-16072	WebGPU		▌	2018-17470
Media Stream	Overflow	▌	2019-5824			▌	2018-17470
	Spoof	▌	2018-6103			▌	2018-6073
MediaRecorder	UAF	■	2018-18340			▌	2018-6154
Navigator	Incorrect security	▌	2018-6041			▌	2019-5770
Payment	Insufficient policy	▌	2018-20071	WebRTC	Incorrect security	▌	2018-6130
		▌	2019-13763		OOB read	▌	2018-16083
	UAF	▌	2019-5828			▌	2018-6129
PDF	Inappropriate implementation	▌	2018-20065		Others	▌	2018-6132
	Insufficient validation	■	2016-10403		Overflow	▌	2018-6156
	OOB read	▌	2018-16076		Type Confusion	▌	2018-6157
	OOB write	■	2017-5133		UAF	■	2019-5760
		▌	2018-6144			▌	2018-16071
	Others	■	2019-13679	XMLHttpRequest	Incorrect security	■	2019-5832