ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Arcanum: Detecting and Evaluating the Privacy Risks of Browser Extensions on Web Pages and Web Content

Qinge Xie
Georgia Institute of Technology

Manoj Vignesh Kasi Murali
Georgia Institute of Technology

Paul Pearce
Georgia Institute of Technology

Frank Li
Georgia Institute of Technology

Abstract

Modern web browsers support rich extension ecosystems that provide users with customized and flexible browsing experiences. Unfortunately, the flexibility of extensions also introduces the potential for abuse, as an extension with sufficient permissions can access and surreptitiously leak sensitive and private browsing data to the extension's authors or third parties. Prior work has explored such extension behavior, but has been limited largely to meta-data about browsing rather than the contents of web pages, and is also based on older versions of browsers, web standards, and APIs, precluding its use for analysis in a modern setting.

In this work, we develop Arcanum, a dynamic taint tracking system for modern Chrome extensions designed to monitor the flow of user content from web pages. Arcanum defines a variety of taint sources and sinks, allowing researchers to taint specific parts of pages at runtime via JavaScript, and works on modern extension APIs, JavaScript APIs, and versions of Chromium. We deploy Arcanum to test all functional extensions currently in the Chrome Web Store for the automated exfiltration of user data across seven sensitive websites: Amazon, Facebook, Gmail, Instagram, LinkedIn, Outlook, and PayPal. We observe significant privacy risks across thousands of extensions, including hundreds of extensions automatically extracting user content from within web pages, impacting millions of users. Our findings demonstrate the importance of user content within web pages, and the need for stricter privacy controls on extensions.

1 Introduction

Web browsers manage some of the most sensitive user data. From emails, to banking information, to medical records, to social media, web pages display private information and users rely on web browsers to ensure information remains secure within their devices, and available only to the correct parties.

At the same time, browser extensions serve a fundamental role in the web ecosystem. Used by millions of users, extensions enhance browser functionality with expressive, powerful features. Google's Chrome Web Store hosts over 100K extensions with billions of total installs [11].

Unfortunately, the same access and capabilities that extensions rely on to enrich the web browsing experience can also be abused to harm user privacy; extensions can collect sensitive user data at scale, potentially without users' knowledge or explicit consent. Even in cases where data collection is benign and necessary for legitimate extension functionality, it introduces privacy risks as sensitive user data can be transmitted and stored by a third party, which may further share the data or possibly leak the data during a data breach.

The intersection of the sensitive nature of some websites and the powerful nature of extensions creates a core privacy conflict: how can we secure *some* websites and types of content when third-party code has significant access to that information. One mechanism deployed by browsers requires extension developers to explicitly document what kinds of content can be accessed, and on what sites. Unfortunately, these permissions are coarse-grained; a simple extension that changes page colors requires the ability to interact with all content on the page. Such construction still affords abuse.

This avenue of privacy problems has not gone unnoticed. Prior work such as Mystique [4] sought to develop an analysis framework for Chromium—the most popular browser platform, underpinning more than 68% of all users' browsing [58]—to explore how extensions access some kinds of user information such as URLs and passwords. Unfortunately, the architecture of modern browsers, the expressiveness of extension APIs, and the web itself have advanced since such tools were developed, making them incapable of operating in a modern context. Orthogonal to prior work, another remaining question is the role that extensions play in the collection of web page *content*, and once they access that information, how it is processed, stored, and potentially exfiltrated.

To address these limitations, we present Arcanum, a dynamic taint tracking system for Chromium designed to track sensitive user content on modern web pages and extensions. The key distinctions of Arcanum from prior systems are its ability to track user data from within web pages, operate on the modern browser architecture, and support taint propagation across a more comprehensive set of browser, web, and JavaScript (JS) APIs used by extensions (including new APIs as well as important ones not previously accounted for, such as the Fetch network request, and data encoding/encryption).

Arcanum understands a diverse set of data sources ranging from meta-data, to content DOM elements, location information, history data, and cookies. From these sources, Arcanum is able to track data flow to a variety of exit sinks, including all forms of web requests and storage APIs. Arcanum does this by instrumenting both the Blink browser engine and the V8 JS engine to mark taint for sensitive data objects (including those returned by sensitive APIs) and comprehensively propagate taint across data manipulation functions. A key feature of Arcanum is allowing researchers to instrument *specific* web page elements as tainted at runtime via JS DOM annotations. This allows Arcanum to not only track how extensions use, manipulate, store, or exfiltrate *specific* data objects, but also reduce flagging of non-sensitive data, such as page colors.

We deploy Arcanum at scale to study all 113,099 functional extensions in the Chrome Web Store. We test each extension against seven privacy-critical sites covering a diverse set of categories: Amazon, Facebook, Gmail, Instagram, LinkedIn, Outlook, and PayPal.

We observe that the automated collection of potentially private data is pervasive; we uncover 3,028 extensions collecting sensitive user data, impacting up to 144M users. Of these, the super majority exfiltrate the data, with a minority storing it locally. We also observe the collection of sensitive user data from within web page *content* by 202 extensions, potentially affecting up to 300K+ users, which has not been previously investigated. The collected information includes the contents of emails, private social media profiles and activity, banking information, and professional networks. Further, all sites tested are impacted by thousands of extensions.

In summary, our contributions include:

- Identifying that more than 58% of today's extensions cannot be analyzed by the existing extension analysis system [4].
- Designing and implementing Arcanum, a dynamic taint tracking system driven by runtime annotations, for modern Chromium browsers.
- Performing a study of *all* functional extensions from the Chrome Web Store, across 7 popular and privacy-critical sites covering email, banking, and social media. We present analysis across extensions, sites, taint sources, and sinks.
- Finding 3,028 extensions automatically collect private user data across sites, impacting up to 144M users; the super majority of these extensions exfiltrate the data off-device.
- Uncovering 202 extensions collected detailed user *content* from web pages, impacting as many as 300K+ users.

We open-source Arcanum at `https://github.com/BEESLab/Arcanum/` to support future research. We have also shared our results with Google and the affected sites.

2 Background and Motivation

In this section, we provide background on Chrome browser extensions. We also discuss related prior work and their limitations, motivating our system and study.

2.1 Chrome Browser Extensions

Extensions expand upon browser features and functionality. Chrome extensions are composed of several core components.

Manifest File. Every extension has a `manifest.json` file specifying an extension's metadata and configuration, such as the permissions required by the extension. In December 2020 [12], Chrome 88 rolled out a new version of its extension platform, called Manifest Version 3 (MV3) [10]. Compared to the previous version MV2, MV3 includes new features and functionality, as well as changes to existing ones. The Chrome Web Store no longer accepts new MV2 extensions. While existing MV2 extensions remain in the Web Store, Google will begin disabling MV2 extensions in pre-stable versions of Chrome in June 2024 [21]. To migrate to MV3, extensions must at a minimum update their manifest file.

Background Scripts. Background scripts (or service workers in MV3) provide the long-term state and functionality of an extension, independent of browser windows/tabs. Background scripts can use Extension APIs (e.g., `chrome.history`) if the necessary permissions are granted to the extension (e.g., the "`history`" permission).

Content Scripts. Background scripts cannot directly access web pages in browser windows/tabs. Instead, extensions can inject content scripts into a window/tab, which runs within the page's context and thus can access the page's DOM interface. In order to inject into a page, extensions must define host permissions in the manifest file (or request it via the "`activeTab`" permission). Content scripts can be injected by an extension either through declarations or programmatic methods:

- Extension manifest files can statically declare content scripts to always inject into web pages with URLs that match specified patterns (including wildcard patterns). Starting with Chrome 96, content scripts can also be dynamically declared, registering content scripts using the `chrome.scripting.registerContentScripts` API and determining injection at runtime.
- Extensions can also programmatically inject a content script through `chrome.tabs.executeScript` (MV2) or `chrome.scripting.executeScript` (MV3).

Content scripts can communicate with the extension background script through message passing APIs, such as using `tabs.sendMessage` and `runtime.postMessage`.

2.2 Prior Browser Extension Privacy Work

The security and privacy research community has broadly investigated browser extensions, such as by assessing extension fingerprinting [23, 28, 33] and discovering vulnerable data flows in extensions [20, 31]. In this work, we specifically focus on studying privacy leakage via browser extension behaviors, which prior work has investigated through differing methods. The types of user information investigated by prior work include those provided by browser or web API calls [3, 4, 17, 20, 32, 67, 68], Document properties (e.g., cookies) [4, 17], and specific HTML element types (e.g., password-typed fields in HTML forms) [4, 22]. Note that these information sources are generically available across all sites, affording straightforward evaluation at scale. However, prior work has not investigated sensitive user data from *within* web pages themselves, such as account profiles, user emails and posts, and personal images. In our work, we seek to expand beyond the user information considered by prior work and account for the wealth of user data within web pages.

To evaluate extension privacy leakage, some works [32, 67] have applied network monitoring and analysis, searching for user data within network requests generated by extensions. The network vantage point is ultimately limited though, as it struggles to identify encoded, encrypted, or otherwise obfuscated user data, and provides limited visibility into how the data is collected and exfiltrated. To provide deeper visibility into extension activity, both static analysis [20, 22] and dynamic taint tracking [4, 17] approaches have been used. Extension static analysis by itself cannot comprehensively identify data leakage flows though, as extension behavior is often heavily dependent on dynamic values that can only be determined at runtime on real-world sites. Thus, in this study, we adopt the dynamic taint tracking approach, of instrumenting a browser to trace how sensitive tainted data may eventually reach exfiltration points. Most similar to our approach is Dhawan et al. [17], who implemented taint tracking for Firefox, and Chen and Kapravelos [4], who developed taint tracking for Chromium. However, in our work, we develop a new browser taint system, Arcanum, to overcome critical limitations with these prior systems, as detailed next.

2.3 Motivation for Arcanum

Our study focuses on extensions for Chromium, the most popular browser platform [58]. While prior work [4] developed extension taint tracking for Chromium, we identified critical limitations necessitating a new system for modern Chromium.

Mystique [4] was developed in 2018, built upon Chromium version 54 released in November 2016. Modern Chromium has significantly evolved since this browser version though. Beyond supporting additional features and APIs, the internal browser architecture has been been substantially updated, including redesigning the JS engine. As a consequence, modern websites and extensions do not function correctly on such an outdated version of Chromium. The browser changes have also been significant enough that Mystique cannot be easily ported to the new version of Chromium. We identify the following browser changes that drive the need for a new system:

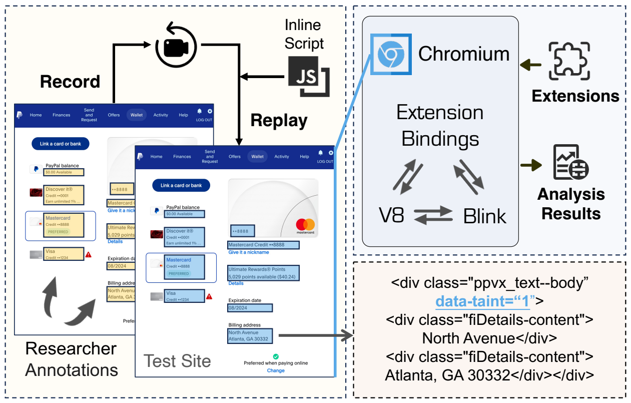
Changes to the V8 JS Engine. Chromium uses the V8 JS engine, which in 2017 migrated to an entirely new JS execution pipeline (from Full-codegen/Crankshaft to Ignition/TurboFan) [64]. V8 has further updated its internal implementation, such as optimizing its garbage collection in 2018 [59, 60] and applying pointer compression to reduce memory consumption in 2020 [62]. As a consequence, Mystique's design cannot be directly ported to modern Chromium versions, as the JS engine architecture is distinct. Instead, a Chromium taint tracking system specific to the modern V8 architecture is needed.

Native Code Data Flow. Prior versions of Chromium implemented core browser functionality in JS, including built-in functions and the extension bindings system that supports extension APIs [14]. These features provide both potential taint sources as well as methods that could propagate tainted user data. However, modern Chromium has migrated these functionalities to native C++ for security and performance reasons, as discussed in Section 3.2.3. Thus, taint tracking on current Chromium requires tracking taint through both JS and native code, necessitating an updated taint tracking system.

Manifest Version 3. As described in Section 2.1, Chrome released a new version of the extension platform, MV3, in 2020. MV3 introduces new APIs and features; MV3 extensions cannot operate on older browser versions that do not support it. Mystique, built upon Chromium 54, cannot run MV3 extensions.

To identify the extent of this limitation, we downloaded all extensions in the Chrome Web Store (leveraging the Store sitemap [7]) in both August 2022 and August 2023. In 2022 we observed 20.65% of extensions (out of 118,655) utilized MV3, while in 2023, the prevalence of MV3 had increased to 58.89% of extensions (out of 114,714). Thus, any attempt to utilize prior taint tracking systems on modern extensions precludes an ever-increasing majority of extensions. It is vital that a taint tracking system supports MV3, especially given that the Web Store no longer accepts new MV2 extensions.

Broken Websites. Since Chromium 54 (which Mystique used), numerous new JS expressions and operators have been added to Chromium. Many of these features have been adopted by websites. For example, the LinkedIn website automatically loads a JS snippet that uses the Nullish coalescing operator, a feature only supported in Chrome 80+. Furthermore, the Spread syntax in object literals is widely used today [1], a feature only supported in Chrome 60+. We identified early in our study that many websites displayed fatal errors when visited in an older browser (e.g., LinkedIn cannot load in Chromium 54, as its core JS scripts relied on missing operators and errored). Thus, not only will an outdated browser impact taint tracking, it inhibits the analysis of modern websites.



```
<div class="ppvx_text--body"
data-taint="1">
<div class="fiDetails-content">
North Avenue</div>
<div class="fiDetails-content">
Atlanta, GA 30332</div></div>
```

Figure 1: Overview of Arcanum. Researchers identify target sites and annotate privacy-sensitive data on web pages. Arcanum then replays content across all extensions in a given dataset, using its instrumented version of Chromium, producing detailed taint source and sink logs per extension and site.

3 System Design

We now present Arcanum, a browser framework that utilizes dynamic taint tracking to detect and analyze the usage of privacy-sensitive data by browser extensions. Figure 1 shows an overview of Arcanum's architecture.

Arcanum allows researchers to understand if specific elements of a web page's content or meta-data are processed, stored, or exfiltrated by extensions. Arcanum provides a variety of privacy-sensitive taint sources (Section 3.1) and sinks (Section 3.3), and monitors the flow of tainted data through its taint propagation engine (Section 3.2), including in the extension's context. Unique to Arcanum is its support for tainting sensitive user data within web pages, through researcher-provided annotations. To evaluate extensions at scale, Arcanum records and replays annotated webpages across all browser extensions (Section 3.5), revealing how data is processed and flows.

3.1 Taint Sources

To ensure the comprehensiveness of Arcanum in tracking data flows containing sensitive information, we surveyed all Chrome Extension APIs [6] and Web APIs [19], and list taint sources supported by Arcanum in Table 1. Note that there are other Chrome Extension/Web APIs supported by Arcanum in the taint propagation process but are not taint sources, such as Web Crypto APIs, which will be discussed in Section 3.2. Overall, our sources can be categorized into three groups:

- Chrome APIs: Extensions can utilize Chrome Extension APIs to directly query for privacy-sensitive information (e.g., `Chrome.history.getVisits`) or dispatch events to notify the extension when specific browser actions are triggered and subsequently return privacy-sensitive data (e.g., `Chrome.tabs.onUpdate`). Arcanum includes the `Chrome.cookies` API as a taint source that can query websites' cookies, which was not considered in prior work. Arcanum also supports information fields that only exist in newer Chrome versions (e.g., "PendingUrl", "initiator", and "ip"). We also further consider the "title" field for the history and tabs API, which can include sensitive information.
- Web APIs: While prior work [4] did not recognize any Web APIs as taint sources, extensions can leverage Web APIs to retrieve privacy-sensitive information. Arcanum thus supports the History, Geolocation and User-Agent APIs.
- DOM Elements: Extensions can access users' private information through the DOM interface. For example, the DOM property `document.title` gives the title of the page, which is supported by Arcanum. Notably, Arcanum allows researchers to mark custom DOM elements that contain sensitive information as taint sources (discussed subsequently).

Notably, while prior work focused on browser APIs as taint sources [3, 4, 68], Arcanum expands beyond prior work to also encapsulate webpage-specific taint sources. Arcanum is designed to also be easily extensible, should additional taint sources be desired in the future (e.g., new browser APIs or other site-specific tainted data). This is accomplished by Arcanum separately handling native data flows in Chrome's extension bindings system (Section 3.2).

DOM Tainting. Arcanum allows researchers to generate custom annotations to taint specific DOM elements on a per-web page basis, enabling investigation of exactly what information is consumed by extensions. This flexibility allows researchers to focus on the parts of the web page that are sensitive, and ignore portions that are not, e.g., we can taint the content of emails, while ignoring colors or themes (common extension behaviors). This custom annotation method is a trade-off; each web page we are interested in must be annotated, but in exchange we get fine-grained information about extension behavior, and reduced false positives of privacy concerns.

For each target web page, researchers identify DOM elements that include sensitive information and label them as taint sources via adding a "`data-taint`" attribute. The HTML `data-*` attribute [18] can be used to store custom data private to the page or application. We designate these element nodes as *labeled* nodes. We modified Blink so that when the "`data-taint`" attribute of an element node is set, Blink traverses all of its descendant nodes, marking the content of all text and CDATA nodes as tainted.

We briefly note that our taint annotation strategy is resistant to evasion. Once an element is tainted (i.e., by a researcher's annotations), its taint status is maintained by the browser internals and cannot be modified via JavaScript (i.e., extension content scripts), even if removing the "`data-taint`" attribute later on. In Section 3.4, we discuss our method for tainting elements prior to extension content script execution.

Accessing Tainted DOM Elements. Extensions can use different HTML element properties to retrieve the data content of

Category	**Taint Source**	**Permission**
DOM custom elements	**innerText/outerText, innerHTML/outerHTML, textContent, wholeText, nodeValue, jQuery text(), etc.**	Content script injection
DOM location	Href, Protocol, Host, Hostname, Pathname, Search, Origin, Hash	Content script injection
DOM property	**URL**, **Domain**, **Title**, Cookie	Content script injection
DOM Input Element	<input type="password">	Content script injection
Chrome.history API	URL, **Title**	"`history`" permission
Chrome.tabs API	URL, **PendingUrl**, **Title**	"`tabs`" permission
Chrome.cookies API	**Domain, Path, Name, Value**	"`cookies`" permission
Chrome.webNavigation API	URL	"`webNavigation`" permission
Chrome.webRequest API	URL, **Initiator**, **IP address**, Cookies in request/response headers	"`webRequest`" permission
History Web API	**URL**	-
Geolocation Web API	**Position**	-
User-Agent Client Hints	**Brands, Platform, Architecture, Model, PlatformVersion, UaFullVersion**	-

Table 1: Taint sources supported by Arcanum. Sources in bold have not been considered by prior work [4].

a node within the DOM, such as "`innerText`", "`innerHTML`" and "`NodeValue`". As the `innerText`/`outerText` property recursively collects text nodes from all child elements within the specified element, we flag the resulting value as tainted if any of the recursively retrieved text nodes are tainted. A similar rule applies to `innerHTML`/`outerHTML`; we also mark the return value of `innerHTML`/`outerHTML` as tainted if any descendant nodes or ancestor nodes of the retrieved node have the "`data-taint`" attribute. Arcanum handles the `wholeText` and `TextContent` properties in the same way as `innerText`/`outerText`. The `nodeValue` property (used by the common jQuery `text()` method) returns the content of a text node (or content of the CDATA section of a CDATA node), and is tainted if the text node is tainted. We also separately taint the `value` property of HTMLTextAreaElement and the `text` property of HTMLTitleElement. We further notice that Blink uses the `StringBuilder` class to aggregate texts within various HTML elements, such as the <img> `title` attribute and the `text` attribute of HTMLOptionElement. Thus, we modify the `StringBuilder` class so that whenever a tainted string is appended to a `StringBuilder` object, we mark the aggregated return values as tainted (i.e., the return value of the `StringBuilder.ToString()` function).

3.2 Taint Propagation Engine

To track the flow of tainted information across the extension's JS execution, Arcanum instruments the V8 engine that Chromium uses to parse and execute JS. V8 parses JS source code into an abstract syntax tree (AST). Arcanum first marks AST nodes as tainted when the corresponding concrete runtime objects (e.g., string objects) are tainted. Then for each individual JS function that is in the Extension context (discussed below), a data flow graph (DFG) is constructed from the AST to process taint propagation that starts from the tainted AST nodes. When this propagation occurs, the corresponding runtime objects linked to these tainted AST nodes are also marked as tainted. This method is consistent with prior work [4], although implemented on the modern Chromium V8 engine. To our knowledge, there are currently no plans for major architectural changes to the V8 engine, indicating that Arcanum should be applicable for the foreseeable future.

3.2.1 Tracking Extension Context

Arcanum exclusively monitors JS execution within the Extension context, which distinguishes it from scripts initiated by the website page itself. V8 uses a `Context` object to represent a JS execution environment, enabling concurrent execution of distinct JS applications within a single V8 instance. For MV2 extensions, this `Context` object can be used to identify JS code that is from the extension's content script or background page. We use this `Context` object to identify and restrict taint propagation to JS code that belongs to an extension, similar to prior work [4, 17].

However, this strategy does not fully translate to MV3, where service workers replace extension background pages. While the `Context` of content scripts and background scripts in extensions are initiated by Blink, service workers are managed by Chrome's extension bindings system. For each extension, the bindings system installs extension API bindings before a service worker starts evaluating its top-level script. Arcanum modifies the installation function to mark the service worker `Context` before any service worker starts execution, allowing us to also identify service worker JS code.

3.2.2 JavaScript Data Flows

We first discuss how Arcanum tracks the flow of tainted data purely within JS execution.

Explicit Flows. While the previous Full-codegen compiler in V8 directly generated unoptimized machine code, the Ignition interpreter generates V8 bytecode from the AST. These bytecode instructions are then interpreted by the TurboFan compiler to generate optimized machine code. Arcanum modifies the Ignition interpreter in V8 to construct the DFG and propagate taint status for AST nodes and runtime objects. The V8 bytecode instructions can be broadly categorized into three categories: 1) assignment operations, 2) arithmetic and logic operations, and 3) control dependencies. Arcanum handles assignment operations by tainting the left-hand side (LHS) if the right-hand side (RHS) of the assignment is tainted. Additionally, when the RHS contains arithmetic and logic operations, Arcanum considers the expression results as tainted if any of the arguments within the expression are tainted. Lastly, in the context of control dependencies (such as `switch-case`, `if-else`, and `do-while` loops), if any arguments within the conditional expression of a control structure are tainted, the LHS in every assignment operation contained within the control structure is also tainted [4,17,65]. Note that Arcanum supports all current JS operators, including those introduced after prior Chromium-based tainting systems [4] (e.g., `await`).

Implicit Flows. In addition to explicit data flows, we must also consider implicit data flows in JS. For example, as we construct the DFG at a per-function granularity, we must also ensure the propagation of taint status across function calls. V8 treats the JS global scope as an anonymous function. Arcanum taints function return values if the return statement expression is tainted via propagation during explicit data flows (within a function). As these tainted function return values reside on the RHS, they subsequently propagate to the LHS through assignment operations. Literal creation for compound types is another scenario that involves implicit JS data flows. For instance, when an extension creates an array literal: `a = [x, x+"pad", "str"]` with a tainted string `x`, it is equivalent to assigning `x` to `a[0]` and `x+"pad"` to `a[1]`. Thus, Arcanum propagates taint to `a[0]` and `a[1]`, but not `a[2]`.

3.2.3 Native Code Data Flows

As discuss in Section 2.3, prior Chromium-based tainting systems [4] predate modern Chromium's migration of many functions previously implemented in JS to native code. In earlier Chromium versions, many browser internal functions were implemented in JS, and thus taint propagation even through browser internal JS was directly handled by the same process propagating taint for explicit JS flows, as described in Section 3.2.2. However, with modern Chromium's migration of these functions to native code, Arcanum requires a distinct approach that makes all possible data flow paths through native code taint-aware.

Built-in Functions. In V8, built-ins implement core functions executed at runtime. The prototypes of JS objects (e.g., `String.prototype.substring()` and `Array.prototype.join()`) are implemented as built-ins, which must be accounted for during taint propagation. In earlier V8 versions, built-ins were widely implemented in JS. However, modern V8 has largely migrated built-ins to CodeStubAssembler, Torque, and native C++ code for performance and reliability reasons. V8's CodeStubAssembler (CSA) [61] is a custom assembler language that provides low-level functionality, while Torque is a wrapper over CSA that simplifies V8 code development. A built-in may involve Torque, CodeStubAssembler, and C++ functions, which requires Arcanum to properly propagate taint across all involved operations. For example, the Torque built-in `String.prototype.slice()` uses a CSA built-in for string addition, which further invokes a C++ runtime function for handling string addition.

We surveyed all built-ins [63], and Arcanum modifies the functions to propagate taint, including those associated with String, Array, RegExp, JSON, ArrayBuffer, and TypedArray objects. Note that we exclude prototypes that do not involve taint propagation, such as `String.prototype.indexOf()`.

Extension Bindings System. The extension bindings system supports browser APIs, including all Extension APIs, which are critical for taint tracking. In early Chromium versions, this system was implemented primarily in JS [14], due to ease of development and the limited interactions between Blink and V8. However, the modern bindings system has transitioned to natively-implemented bindings. Arcanum modifies the native implementation to handle tainted data flows, and also supports MV3 features in the bindings system (described below).

Promises. In MV2, Extension API methods can input a callback function to process results, and we directly taint any privacy-sensitive information passed to the callback function in the extension bindings system. However in MV3, extension API methods can either use callback functions or return a Promise, where a Promise is an object that serves as a proxy or placeholder for the value eventually returned by the asynchronous method. We cannot directly taint privacy-sensitive information when Promises are created in the bindings system, and instead our implementation dynamically taints information in V8 when the Promise is finally "fulfilled" at runtime.

ExecuteScript. Arcanum also propagates taint when a JS function is compiled from a tainted string. In MV2, the tainted string may be compiled by the `eval` function and the `tabs.executeScript` Chrome API. Arcanum taints all LHS targets of assignment expressions if the code string is tainted, in a manner akin to [65]. MV3 additionally introduces the `chrome.scripting.executeScript` API. Unlike the tabs API, which solely accepts static string codes or files as input, this new API allows the direct inclusion of JS functions and associated arguments to be passed to the included functions. We do not address the scenario in which the injected scripts are sourced from files, as we do not expect a tainted value to be embedded in a static JS script and subsequently injected programmatically into a web page. In the case of injected scripts containing tainted strings, Arcanum handles them similarly as with `eval`. For JS function injection, Arcanum taints all LHS targets of assignment expressions including the function return values, if any specified input arguments are tainted.

Binary Data Buffers. Another important native data flow that Arcanum augments beyond prior work is handling binary data buffers. Previous work was constrained to taint propagation only among strings, handling string-to-string [24, 25] and/or from string to an object containing a string (e.g., an array of strings) [4, 17, 65]. However, extensions may convert strings to and from binary data buffers, such as when encoding or encrypting strings, and web request APIs (e.g., Fetch) support sending binary data buffers.

Such string encoding methods are not implemented by V8; rather, they are components of the Web APIs handled by Blink. Arcanum modifies Blink to propagate taint status for binary data buffers as well. Specifically, Arcanum tracks propagation between: 1) strings to binary data buffers, such as with `TextEncoder.encode()`, 2) binary data buffers to strings, such as `TextEncoder.decode()`, and 3) between binary data buffers, like with `SubtleCrypto.encrypt()`.

In Blink, a raw binary data buffer is represented using an `ArrayBuffer` object. Additionally, an `ArrayBufferView` object serves as a higher-level abstraction that offers a structured view on an `ArrayBuffer`, providing methods for manipulating the binary data contained within the `ArrayBuffer`. In contrast to Blink strings, which are exposed as the string type in V8, objects like `ArrayBuffer` and `ArrayBufferView` inherit from the `ScriptWrappable` class in Blink, which serves to specify type information when these objects are exposed in V8. Thus, in Arcanum, we address the conversions from `ScriptWrappable` objects to V8 objects and vice versa, specifically when the types are `ArrayBuffer` and `ArrayBufferView`. Since `ArrayBufferView` objects are exposed as `TypedArray` in JS, we modified all built-in functions that implement the prototypes of `ArrayBuffer` and `TypedArray` to ensure the correct propagation of taint, which include prototypes like `ArrayBuffer.prototype.slice()` and `TypedArray.prototype.subarray()`.

Arcanum propagates the tainted positional information for all string-to-string operations, but also between binary data buffers as well. For cryptographic functions, such as `SubtleCrypto.encrypt()`, we fundamentally cannot track which output bytes depend on tainted input bytes. Thus, Arcanum marks all bytes in the output binary data buffer of cryptographic functions as tainted if the input is tainted. We handle Base64 encoding similarly.

3.3 Taint Sinks

Arcanum tracks if tainted data propagates to taint sinks, where it is potentially exfiltrated by an extension. These sinks, as shown in Table 2, can be grouped into: web requests, extension-injected DOM elements, and persistent storage.

Category	Taint Sink
Web Request	**Fetch**
Web Request	XMLHttpRequest
Web Request	WebSocket
Web Request	**Beacon**
DOM	DOM elements injection
Storage	Chrome.storage API
Storage	Web Storage API
Storage	**IndexDB**

Table 2: Taint sinks supported by Arcanum. Sinks in bold have not been considered by prior work [4].

- **Web Requests.** Extensions can leverage web request APIs to transmit sensitive user information externally. Arcanum tracks whether any tainted value is sent as any part of a network request, including request headers, URL parameters, or request bodies. These web request APIs include XMLHttpRequest, WebSocket, Fetch and Beacon. Prior work [4] was limited to taint tracking for text in the request body; Arcanum extends its support to cover URLSearchParams, FormData, ArrayBuffer, and ArrayBufferView as request body formats (as discussed in Section 3.2). We also note that prior work [4] did not account for Fetch as a taint sink, which has been supported since Chrome 42 in 2015. Also now in MV3, XMLHttpRequest can no longer be invoked from a service worker (the background script) [8], forcing modern extensions to migrate from XMLHttpRequest to Fetch. Extending beyond prior work, we also include the Beacon API that is used to send an asynchronous request to a server, which could also be a data exfiltration channel.
- **Extension-injected DOM Elements.** Similar to prior work [4], Arcanum inspects DOM elements injected by extensions, in order to determine if their `src` attribute values contain tainted values. As browsers automatically fetch content from the `src` attribute specified URL, extensions can embed sensitive data within the URL to leak data.
- **Persistent Storage.** Extensions can locally store privacy-sensitive information. This presents two potential privacy threats: 1) an extension could potentially exfiltrate from storage immediately, or could do so at some later time; 2) buggy or poorly implemented extensions may be vulnerable to attack by websites, and any information they store locally could be exfiltrated by those websites [20]. Arcanum inspects if any tainted values are stored by extensions, accounting for Chrome.storage, Web storage, and IndexDB (which was not considered by prior work).

3.4 Delayed Content Script Injection

Tainted elements on a web page may not load immediately (such as if dynamically loaded). To maximize the likelihood that we detect an extension's access to tainted data, we configure Chromium to delay extension execution until after web pages (and tainted elements) are fully loaded. To do so, we configure the content script `run-at` parameter [9] in Blink that controls when an extension's content script is injected into a page (regardless of static or dynamic injection).

We enforce that the `run-at` parameter is always set to `document_idle` even if an extension configuration specifies otherwise, such as `document_start` or `document_end`. Blink currently reaches the idle status when the earliest of two states arises: 1) after fully loading the document and all subresources, or 2) 200 milliseconds after fully loading the document (but some subresources may still be loading). We introduce a forced delay after Blink reaches the idle status, specific to the load times of a target page. We configure this delay as an Arcanum input parameter, allowing it to be tuned without rebuilding Chromium. Thus, a user can configure this delay to ensure that a target web page will fully load before an extension injects its content script, such as if the initial page load involves some animation (e.g., on Gmail and LinkedIn).

3.5 Web Page Replay

When evaluating extension behavior on a web page, we opt to record that web page and replay it across the extensions evaluated, using WprGo [2]. Popular websites are known to actively combat automated activity on their pages [26, 27]. Thus, frequent page load activities likely lead to forced account logout or account suspension/termination. Furthermore, we observed that some sites periodically mutate to combat automation/scraping, such as by randomizing or periodically changing HTML element IDs or CSS class names. We found that Facebook pages use randomly-generated CSS class names that change every few weeks, while DOM element IDs in LinkedIn pages change for each page load. Using replayed web pages allows for consistent evaluation of these pages over time and over extensions. In addition, replaying web pages reduces the load on investigated websites (as discussed further in Section 4.1). While we utilize replayed pages both for our system's design and subsequent evaluation, we note that Arcanum itself does not require replay; Arcanum could be run on live web pages.

4 Evaluation

We now describe our deployment of Arcanum across all Chrome extensions and seven websites rich with user data. Compared to prior work [4], our evaluation is not only for extensions on modern Chromium, but also provides more comprehensive and finer-grained analysis of data exfiltration.

4.1 Experimental Setup

Implementation. Our prototype implementation of Arcanum is built on Chromium Browser version 108.0.5359.71 (released in November, 2022) [5]. We chose this specific version as it supports both MV2 and MV3 extensions, whereas the latest versions of Chromium cannot run MV2 extensions [13]. While the Chrome Web Store stopped accepting new MV2 extensions, we found that nearly half of the existing extensions are still using MV2 (Section 2.3). This Chromium version and Arcanum fully support modern MV3 extensions (as of March 2024), and we have reviewed browser changes since this version and did not find any changes that would impact our taint sources, sinks, and propagation.

Target Sites. Previously, individual cases of browser extensions collecting user data from social network sites have been documented [15, 16]. Based on these observations and prior work, we experiment with seven popular sites rich with user content. The sites and specific pages we explore are Amazon (profile/address information), Facebook (wall/post information), Gmail (inbox), Instagram (profile information), LinkedIn (profile information), Outlook (inbox), and PayPal (credit card information). Table 3 lists the details of tainted information on each page. These sites cover social networking, email services, e-commerce, and financial services.

For each site, we manually created test accounts populated with fictitious but realistic data, providing a wealth of information that an extension may potentially access. We manually identified the HTML elements containing the user data on each web page. When recording and replaying a page (Section 3.5), we inject our own custom JS script that immediately taints the user data HTML elements (with the `data-taint` attribute, Section 3.1). We taint elements with JS as we observed that many elements are dynamically constructed, and thus cannot be directly tainted in the replayed HTML resources (without modifying web page JS itself, a significantly more brittle and challenging task, especially with JS minification and obfuscation). Note that our extension execution delay (Section 3.4) is configured so that all elements are tainted prior to extension content script injection.

Collecting Extensions. In August 2023, we downloaded all Chrome extensions available in the Chrome Web Store, as listed in the Store's sitemap [7]. After filtering out Themes and Chrome OS Apps, we were able to successfully download 114,714 extensions. To ensure the extensions were still functional, we automated an unmodified Chromium browser (using Selenium WebDriver [29]) and attempted to install each extension. For 1,615 (1.4%) of extensions, we encountered installation errors, primarily due to development bugs (e.g., missing resources) or platform compatibility issues (e.g., being classified as an extension in the Web Store but is actually a Chrome OS App). We exclude these extensions, leaving us with 113,099 extensions to evaluate. We note that all population data given is as of this August 2023 collection.

Running Arcanum. We run each instance of Arcanum in a Docker container on Ubuntu 18.04 (allocated 4 CPUs and 10GB RAM), affording parallel analysis across many

Target page	URL	Title	Tainted information on the page
Amazon-Address	-	-	Name, Physical address (including address and phone number)
Facebook-Profile	User ID	-	Name, Profile, Friend, Post (including Post content, Location, Comments)
Gmail-Inbox	-	Email address	Name, Email address, Last account activity timestamp, Email content (including Email content, Title, Sender, Timestamp)
Instagram-Profile	User ID	User ID, User Name	Name, Profile, Image sources and captions (in <img> `alt` attributes)
LinkedIn-Profile	User ID	User Name	Name, Profile, Friend ("People you may know"), Message
Outlook-Inbox	-	User Name	Name, Email address, Email content (including Email content, Title, Sender, Timestamp)
PayPal-Card	Payment ID	-	PayPal balance, Credit card information (Last 4 digits of the card number, Card issuance institution, Card expiration date), Physical (Billing) address

Table 3: Tainted information on each target page, and privacy-sensitive information within page URLs and titles.

containers. We use Selenium WebDriver [29] to automate Arcanum browser navigation and install extensions. For each extension to be tested on a target page, we launch a fresh instance of our modified Chromium in headful mode (with a fresh user data directory), with the extension pre-installed using the `-load-extension` parameter in Selenium. The Chromium browser is run with an Xvfb [69] virtual display with 1920x1080 resolution, allowing the complete loading of sensitive DOM elements. We use Chromium's `-host-resolver-rules` command line argument to redirect network requests to our replay proxy to replay web resources (Section 3.5). For each extension, we execute it for 60 seconds after the web page has completely loaded. During evaluation, we do not interact with the pages or extensions (e.g., we do not click buttons on the page or deliberately trigger behaviors).

Note that in this evaluation, we *cannot* definitively ascertain the intent behind extensions collecting user data. In many cases, the purpose may be benign. Nonetheless, any extensions detected by Arcanum during our experiment will have automatically gathered sensitive user information after simply installing the extension and visiting a page. Furthermore, even in benign cases, exfiltrating sensitive user data is a privacy risk as the data may be transmitted or stored insecurely, shared with external parties, or compromised in the future.

Taint Log Processing. Arcanum produces detailed taint propagation logs for each extension and web page, documenting what taint sources are accessed, a stack trace of taint propagation, and the taint sinks that are eventually reached. In detail, for a propagation step, Arcanum logs the source JS that propagates the taint status to the destination JS object, along with the JS function and source code position where propagation is triggered. This allows us to trace the path from taint sinks back to their respective taint sources and analyze the privacy-sensitive information accessed by each flagged extension (shortly discuss in Section 4.4), even when taint propagation involves encoding and truncation.

Since Arcanum separately handles native data flows in Blink, it allows us to log the exact methods and positions through which a taint source is accessed (for instance, through the invocation of the chrome.Cookies API), as well as the precise data format that a taint sink is reached (such as a tainted ArrayBuffer object via a Fetch request). We also further modify Blink to enable Arcanum to log particular taint propagation flows we wish to inspect, such as those involving text encoding operations (discuss later in Section 4.6).

We programmatically process logs to trace back from taint sinks reached to the original taint sources. During our log processing, we observed two situations that could result in false positives. First, when an extension is installed, it opens its own web page (e.g., a welcome page or a login page for the website associated with the extension) and accesses a taint source (in this case, it must be a Chrome or web API) that eventually reaches a taint sink. Thus, the tainted information is not directly related to our target web page. Second, when our taint propagation stack trace does not involve extension code or functions, thus indicating that it was not extension behavior that resulted in taint propagation. This second case arises only from our current handling of URLSearchParams in Arcanum now. We filter out both situations when analyzing taint data flows, which affected 232 extensions. Note that we filter data flows only, but if an extension only contains these errant data flows, it is not flagged.

Ethical Considerations. As our study involves large-scale data collection and analysis, we discuss the ethical considerations for each component of this study.

Target Sites. As we analyze numerous extensions on the authenticated web pages of target sites, continuously revisiting these pages on the live website can induce load. Thus, by only visiting the pages once and replaying the captured response, we minimize additional load on a site regardless of the number of extensions evaluated. Furthermore, we use a test account on each site populated with fictitious data, so no real user data or accounts are involved.

Downloading Extensions. We downloaded extensions from the Chrome Web Store serially to rate-limit our requests.

Disclosure. We have shared our results, including specific Chrome extensions found, with Google and all target sites. During the disclosure process, we observed that only Amazon specifically reminds users about extensions' privacy risks

	Total	Amazon	Facebook	Gmail	Instagram	LinkedIn	Outlook	Paypal
Extensions	3,028 (2.68%)	2,048 (1.81%)	1,730 (1.53%)	2,198 (1.94%)	2,067 (1.83%)	2,088 (1.85%)	1,964 (1.74%)	1,943 (1.70%)
Users	144.0M	89.6M	66.3M	86.1M	91.6M	95.7M	85.7M	83.4M

Table 4: Overview of the number of extensions Arcanum identified as having sensitive taint sources flow to an exfiltration or storage taint sink, broken out by website. In total, 113,099 extensions were analyzed. The aggregate number of extension users is also given, however due to potential user overlap, this user count must be taken as an upper bound.

Rank	Extension Name	#Users	Taint Sink(s)	Details	Encoded?
1	Honey: Automatic Coupons & Rewards [47]	10M+	fetch, storage	URL, Timestamp	No
2	Online Security [52]	10M+	fetch	URL	No
3	**Avast SafePrice** [34]	10M+	XMLHttpRequest	URL	Yes
4	Capital One Shopping [38]	8M+	fetch, XMLHttpRequest, storage, DOM	URL, Title, Device	Partial
5	Touch VPN - Secure and Unlimited VPN Proxy [57]	8M+	storage	URL, Country	Partial
6	Avira Browser Safety [35]	6M+	XMLHttpRequest	URL	No
7	**Hola VPN - The Website Unblocker** [46]	6M+	XMLHttpRequest, storage	URL	No
8	**Avira Safe Shopping** [36]	5M+	XMLHttpRequest	URL	No
9	NordVPN - VPN Proxy for Privacy and Security [51]	3M+	fetch, storage	Domain, Timestamp	No
10	QuillBot: AI Grammar and Writing Tool [54]	3M+	fetch, storage	Device	Yes

Table 5: Top 10 extensions, by popularity, flagged across all target sites. Extensions in bold were also identified in prior work [4].

on their customer service page[1], and we identified that sites broadly lack contacts for reporting such privacy-related issues. Thus, in addition to any privacy-related contacts found, we engaged via vulnerability disclosure channels, technical issue reporting, abuse contacts, and personal connections. To date, we have not yet observed corrective actions taken, but we recognize that there may be limited options for such actions (especially by sites), and any actions would need to be carefully considered and executed.

4.2 Results Overview

Table 4 presents our aggregate results. We find that across our seven target websites, 3,028 (2.68%) extensions access a taint source (i.e., sensitive user data) and directly propagate this information to a taint sink (e.g., outbound network request). The aggregate installation base of these extensions is 144M users, which serves as an upper bound on the total user impact, as a single user may install multiple of these extensions.

In total, there are 1,338 extensions (44.2% of flagged extensions) displaying such activity across all of our target sites. A manual investigation identified common libraries being used in many such overlapping cases. For example, 149 extensions use the Sentry Performance Monitoring Library [30], which collects the URL, device, and user agent on every page visited and sends the data to Sentry's servers. Sentry offers other performance monitoring capabilities, and in one instance [55] we observed the extension utilizing their library to send specific web page content containing sensitive user data to the Sentry servers. A previous report [66] identified libraries compromising the privacy of browser extensions, but this is a demonstration of library impact on website content.

These results indicate that extensions pose a significant privacy risk for users, including their sensitive data within web pages. Also, the majority of flagged extensions only operated on certain sites, demonstrating more targeted activity. We note that even if the data collection is benign and necessary for legitimate extension functionality, it introduces privacy risks as sensitive user data is transmitted and stored by a third party (potentially without user awareness, discuss later in Section 4.10), which may further share the data or unintentionally leak it during a data breach.

4.3 Extension Popularity and User Impact

Table 5 lists the top 10 extensions flagged by Arcanum with the most users, what information is accessed, where it flows, and if it is encoded during propagation. We find extensions with millions of users, accessing URL, timestamp, page title, and device information and sending the data over network requests or storing locally (which can still afford data exfiltration or privacy risks, as discussed in Section 3.3). We also find that 4 of the 10 extensions apply some data encoding during taint propagation, justifying Arcanum's design for propagating taint for binary data buffers (Section 3.2).

Figure 2 presents CDFs of extension user installs for our full Chrome extension population, the subset of Chrome extensions that do inject content scripts, and extensions flagged by Arcanum. We see that broadly, the extensions detected by Arcanum are substantially more popular than the general extension population, even constrained to those injecting con-

[1] https://web.archive.org/web/20231126193227/https://www.amazon.com/gp/help/customer/display.html?nodeId=G8V457F4P763VW8D

	Category	Total	Amazon	Facebook	Gmail	Instagram	LinkedIn	Outlook	PayPal
Sources	**Domain**	463 (15.3%)	543 (26.5%)	308 (17.8%)	575 (26.2%)	395 (19.1%)	304 (14.6%)	435 (22.1%)	435 (22.4%)
	URL	1,551 (51.2%)	947 (46.2%)	902 (52.1%)	1,014 (46.1%)	1,112 (53.8%)	1,175 (56.2%)	971 (49.4%)	984 (50.6%)
	Identification	375 (12.4%)	248 (12.1%)	177 (10.2%)	223 (10.1%)	217 (10.5%)	215 (10.3%)	235 (12.0%)	206 (10.6%)
	Title	251 (8.3%)	149 (7.3%)	184 (10.6%)	193 (8.8%)	161 (7.8%)	184 (8.8%)	186 (9.5%)	164 (8.4%)
	Page Content	202 (6.7%)	109 (5.3%)	124 (7.2%)	127 (5.8%)	133 (6.4%)	154 (7.4%)	105 (5.3%)	122 (6.3%)
	Uncategorized	186 (6.1%)	52 (2.5%)	35 (2.0%)	66 (3.0%)	49 (2.4%)	56 (2.7%)	34 (1.7%)	32 (1.6%)
Sinks	**Web Requests**	2064 (68.1%)	1,405 (68.6%)	1,198 (69.2%)	1,613 (73.4%)	1,478 (71.5%)	1448 (69.3%)	1,361 (69.3%)	1,353 (69.7%)
	Storage	362 (12.0%)	318 (15.6%)	249 (14.4%)	312 (14.2%)	306 (14.8%)	349 (16.7%)	296 (15.0%)	312 (16.0%)
	DOM	133 (4.4%)	132 (6.4%)	60 (3.5%)	80 (3.6%)	87 (4.2%)	113 (5.5%)	97 (5.0%)	113 (5.8%)
	Mixed	469 (15.5%)	193 (9.4%)	223 (12.9%)	193 (8.8%)	196 (9.5%)	178 (8.5%)	211 (10.7%)	165 (8.5%)

Table 6: Distribution of privacy-sensitive information that flagged extensions are exfiltrating and the exfiltration methods. If an extension engages with multiple levels of sensitive information, we place it into the most severe category. For instance, if an extension transmits both URLs and page content to a third party, we exclusively categorize it at the page content level.

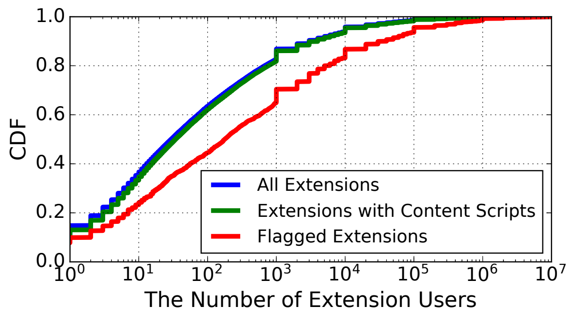


Figure 2: CDFs of extension user populations, considering the extensions flagged by Arcanum (red), the total population of extensions in the Web Store (blue), and the population of extensions that utilize Content Scripts (green). Extensions that Arcanum flagged are significantly more popular than the overall extension population.

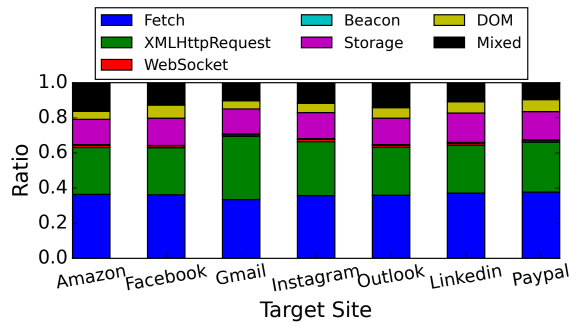


Figure 3: Distribution of the specific sink used by flagged extensions across target sites. "Mixed" represents that flagged extensions reached multiple specific sinks (e.g., "Mixed" extensions include those reaching both Fetch and Storage sinks, but do not include those only reaching Fetch sinks).

tent scripts. For detected extensions, 15% have at least 10K users and 5% have over 100K. Thus, the extensions accessing and exfiltrating user data have an out-sized impact.

4.4 Source and Sink Distributions

Table 6 provides a detailed breakdown of how many extensions activated specific taint sources and sinks, across each target web page. We categorize accessed privacy-sensitive information (sources) into five levels:

- **Domain**: Extensions that only exfiltrate the domain name of the pages that users visit, excluding other URL components.
- **URL**: Extensions that exfiltrate additional URL components (or the full URL) beyond the domain name may expose sensitive information. For instance, user names/IDs are included in the Facebook, Instagram, and LinkedIn URLs, and payment IDs are in some Paypal URLs.
- **Identification**: Extensions that exfiltrate any user identification information, such as IP addresses and device details.
- **Document Title**: Extensions that exfiltrate information from the DOM title. Titles can be more sensitive than URLs, as they can contain confidential information, such as a user's email address on Gmail pages.
- **Page Content**: Extensions that scrape sensitive page content, which may encompass private information, such as a user's physical address and credit card details (Section 4.5).

Of flagged extensions, more than 78.6% exfiltrate privacy-sensitive information beyond just the domain name, with URL-level data being the most common level (51.2%). The number of flagged extensions decreases as the sensitivity level increases, which is consistent across all tested pages. We identified 202 extensions leaking the *contents* of web pages, which has not been previously detected at scale. Of these, 90 extensions are flagged accessing page content across all targets.

For 186 (6.1%) flagged extensions, the specific source content could not be programmatically checked, as we were unable to trace back from taint sinks to their sources. This limitation is due to an implementation issue, and arose when extensions propagate taint using built-in functions in V8, which are currently implemented in Torque

Content Type	Extensions	Max Extension # Users
Name	130	80k+ [50]
Profile	124	300k+ [42]
Email Address	73	10k+ [40]
Location	63	30k+ [43]
Friend	56	30k+ [53]
Credit Card	49	10k+ [40]
Post	49	3k+ [49]
Email Content	46	10k+ [40]
Physical Address	46	10k+ [40]
Comments	39	3k+ [49]
Whole HTML	30	1k+ [41]
Image `alt` Attribute	1	205 [55]
Total	202	300k+ [42]

Table 7: Breakdown of privacy-sensitive information (sources) from within web pages that are exfiltrated by flagged extensions, ordered by the number of flagged extensions. The given number of users is the highest user count of an extension flagged exfiltrating a given content type.

(e.g., `string.prototype.slice`). Our initial Arcanum implementation did not properly log such propagation in Torque (but it propagates taint correctly). Meanwhile, the sink objects recorded in the logs may manifest as a Map or JS Array, and identifying their sources directly necessitates manual efforts. We note these extensions as “Uncategorized”.

Across all target sites, we observe a consistent pattern that taint sinks are primarily web requests (68.1%). Figure 3 provides a more specific sink breakdown beyond the categories of Table 6. Fetch and XMLHttpRequest are the two most prominent sinks that flagged extensions use to exfiltrate privacy-sensitive information. Approximately 15% of extensions reach multiple specific taint sinks. Among these extensions, 87% reach two specific sinks, while 9.3% reach three sinks, and very few extend beyond three sinks (averaging across all target sites). The most common combination among these extensions is Fetch and Storage, followed by Fetch and XMLHttpRequest.

4.5 Web Page Content

We now explore extensions that Arcanum flagged as leaking web page content. Table 7 provides a breakdown of the specific sensitive page content types automatically scraped by extensions, how many extensions scraped each type, and the maximum number of affected users of a flagged extension in that category. In total, 202 extensions exfiltrated one of these sensitive page content types, impacting at least 300K users.

We found that the collection of user’s names and profile information is the most common page content collected, performed by 130 and 124 extensions respectively. The most popular name-collecting extension had over 80K users that collected name and profile information on the LinkedIn page, while an extension with over 300K users collected profile information on the Facebook page. We still observe many extensions collecting other types of user data from web page content, including those with tens of thousands of users.

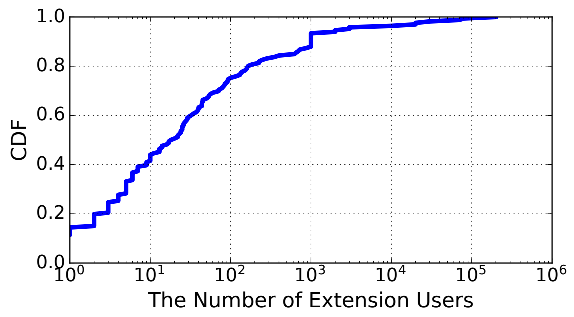


Figure 4: CDF of the number of extension users for extensions collecting sensitive web page user content. The x-axis maximum value is 10^6, differing from Figure 2 and Figure 5.

Figure 4 provides a CDF of the number of users of extensions collecting page contents. A small number of extensions impact many users; less than 10% of extensions impact 1K users or more, with only a few impacting more than 10K. The most impactful extension was installed by more than 300K users. While the total number of users impacted may be limited, the impact can still be significant; these extensions collect credit card information, physical location, personal communication, and more. The prevalence of less popular but privacy-invasive extensions highlights the need for a system like Arcanum that can evaluate extensions at scale.

4.6 Text Encoding

We also discovered 159 extensions transmitting tainted data after using some form of encoding, encryption (at the application layer, not the transport layer), or obfuscation. Such encoding is noteworthy as it can prevent identifying such data exfiltration without dynamic taint tracking, and prior taint tracking systems did not propagate taint through such methods. The three most prevalent forms of data transformations were `TextEncoder.encode[Into]()` (used by 85 extensions), base64 encoding (78 extensions), and `SubtleCrypto.encrypt()` (31 extensions). Extensions utilizing these techniques span the popularity spectrum, as Figure 5 illustrates, impacting as many as 10M users in total.

4.7 Insecure HTTP Usage

We identified 65 extensions exfiltrating sensitive tainted data over network requests sent unencrypted over HTTP (both using XMLHttpRequest and Fetch). This behavior presents a significant added privacy risk to users, as any information

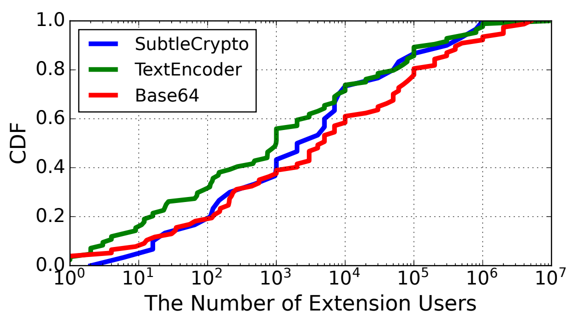


Figure 5: CDFs of the number of extension users for extensions that encode tainted values using 1) `SubtleCrypto`, 2) `TextEncoder`, and 3) Base64 encoding.

transmitted this way would be visible in transit. In the most impactful case, one extension [44] with more than 400K users sent visited URLs in the clear over HTTP. Other extensions [37, 45, 48] sent more sensitive information such as web page content (including for sites where the content is originally served over HTTPS), IP addresses, and page titles.

4.8 Privacy Policies

In the Chrome Web Store, extension developers have the option to indicate their extension's privacy practices in a standardized format. This format includes enumerating the types of information collected by the extension, including personally identifiable information, authentication information, user activity, location data, financial and payment information, personal communications, health information, and website content. In addition, developers can include a link to a web page with further details on how the collected data may be used.

For all extensions flagged by Arcanum, we collected the privacy practices from the Chrome Web Store in October 2023. We observed that 1,575 (52.01%) of flagged extensions did not provide a privacy policy. Of the extensions that specifically exfiltrate web content, 77 (38.12%) offered no privacy policy, including extensions with tens of thousands of users (e.g., [56]). For extensions with privacy policies, 23 (11.39%) gathered website content without listing this data category in their privacy policy, including extensions with more than 80K users (e.g., [50]). These findings suggest that Chrome extensions are currently not heavily incentivized to provide privacy policies, and even if they do, the policies may provide limited accuracy and insights into extension data practices.

4.9 Manual Verification: False Positives

Thus far, all qualitative results and examples presented have been manually verified as leaking sensitive user data. However, while Arcanum accurately detects taint propagation from a source to a sink, it may be possible that the propagated data no longer contains directly sensitive data. To understand the extent to which this may occur and contextualize our results, we conducted an in-depth manual exploration and verification of two groups of 50 extensions, each targeting a different site. We selected the 100 total extensions at random among the flagged dataset. We selected two sites for this exploration: LinkedIn and Paypal. We selected LinkedIn as it was the website that observed the most extensions collecting data from it, and PayPal as it is a completely different class of website that has significant privacy and financial implications.

We observed 3 cases for LinkedIn and 1 case from PayPal where, while taint propagation is correctly inferred, the tainted value may no longer be sensitive. In these cases, tainted values were used in control flow decisions, resulting in tainting of the outputs of the control flow branches. However, the actual exfiltrated data is no longer directly derived from the tainted value. Such behaviors *could* still be privacy violating (e.g., as a covert channel), but it is unclear. As part of this analysis, we also discovered 1 extension for LinkedIn and 2 extensions for PayPal that were sending sensitive values back to LinkedIn and PayPal destinations, respectively. Depending on the specific situation, this may or may not be problematic from a privacy or security perspective. For example, an extension could be taking data from the user's context and exfiltrating it to a context where others can observe it.

While in all cases, Arcanum is accurately detecting taint propagation from a taint source to sink, our analysis here indicates that for a small minority of extensions (approximately 6%), the exfiltrated information may no longer contain clearly sensitive data. Nonetheless, Arcanum serves as an effective platform for analyzing extension data exfiltration behavior.

4.10 Privacy Impact Case Studies

While Arcanum accurately detects cases of automated user data collection, these cases may vary in their privacy implications, from surreptitious exfiltration to collection for legitimate extension functionality (although even if benign, this collection still can entail privacy risks due to the transmission and storage of sensitive user data by third parties, potentially without user awareness, and the third parties may further share the data or leak it unintentionally due to data breaches). To better understand the privacy impact of the flagged extensions, we manually analyzed random samples of extensions (automated analysis is challenging as it requires understanding and comparing extension descriptions and privacy policies with extension behavior).

We do not seek to specifically identify the intent behind an extension's behavior, and instead focus on whether the automated data collection observed by Arcanum is specified in the extension's privacy policy (if existing) or expected based on the extension's description in the Chrome Web Store. In detail, our assessment is as follows:

Random Sample Group	Privacy Policy Practices			Web Store Description		
	#In Policy	#Not in Policy	#No Policy	#Clear	#Vague	#Violative
Web content (20)	8	7	5	3	10	7
All flagged extensions (20)	6	11	3	5	5	10
Total (40)	14 (35.0%)	18 (45.0%)	8 (20.0%)	8 (20.0%)	15 (37.5%)	17 (42.5%)

Table 8: Manual analysis of whether the observed automated data collection is discussed in an extension's privacy policy or Chrome Web Store description, for 40 randomly sampled extensions: 20 extensions sampled from those collecting web page content, and 20 from all flagged extensions by Arcanum.

1) **Privacy Policy.** We characterize whether the observed data collected is listed within the extension's privacy policy. As described in Section 4.8, an extension either does not provide a privacy policy ("No Policy"), provides a policy without listing the specific data categories ("Not in Policy"), or accurately lists the data collection in the privacy policy ("In Policy").
2) **Web Store Description.** We also investigate whether an extension's automated data collection is clearly described on the extension's Web Store page, so that users could reasonably expect the behavior. We label an extension's Web Store description as either "Clear", "Vague", or "Violative". We consider a description clear if it explicitly discusses collecting data automatically for the extension's described functionality. We label a description as vague if automated data collection is not discussed and its functionality could feasibly be implemented client-side rather than requiring data collection, as a user (especially a non-technical one) may not expect automated data collection. We label a description as violative if either: 1) the description claims it does not collected the data; 2) the description clearly states that the extension's functionality would be triggered through certain actions (such as clicking a button), and thus users would not expect data collection to be automatic.

We randomly sampled 20 extensions detected as exfiltrating web page content, and 20 extensions from all flagged extensions. The 40 extensions were then manually analyzed by two researchers independently, who then converged on the final labels as listed in Table 8.

Overall, we found that only a minority of extensions disclosed the data collection within a privacy policy (aligning with our prior observations in Section 4.8) or clearly described the data collection within its Web Store page (with over a third of extension descriptions contradicting the automated data collection). Furthermore, none of the sampled extensions provided both an accurate privacy policy and a clear Web Store description. Thus, for most sampled extensions, users reasonably would not expect the automated data exfiltration based on the extension's available descriptions and privacy policies.

Finally, to further illustrate the privacy risks of the extension behaviors observed, we present case studies of several classes of extensions found by Arcanum.

Detecting Generative AI. We discovered 4 extensions that all purport to detect generative text from systems such as ChatGPT, while exfiltrating web page content to third parties. For example, "DetectGPT - Detect Chat GPT Content" [40] is an extension with 10K+ users that scrapes and exfiltrates *all* tainted user data across all target websites over Fetch network requests. However, its privacy policy states that it does not collect or use user data, and its Web Store description does not discuss data collection and even implies that user action is needed before evaluating page content. We note that there are 13 other extensions with similar behavior purporting to have related AI applications.

Coupons / Financial Benefits. Arcanum found 16 extensions that purport to provide financial benefits such as cashback or comparing offers between web pages. For example, "Sidex Scanner" [56] is one such extension used by 20K+ users, describing itself as offering users on an online store with product offers from other stores. However, it extracts user and profile information from LinkedIn, Facebook, and Instagram and sends the information over XMLHttpRequest requests. The extension does not provide a privacy policy.

Business Contact Services. Arcanum flagged 9 extensions that provide ways for identifying contact information for businesses or professionals. For example, "Mr. E-Find B2B contacts universally" [50] has 80K+ users, and is described as providing contact information for online business profiles upon a user click. However, we observed it automatically collecting and exfiltrating profile and identification information from LinkedIn via Fetch, without listing web content collection in its privacy policy.

Email Assistance. Email Extractor [42] is a popular email assistance extension used by more than 300K users. Its stated purpose is to quickly and easily extract emails from a given web page. Arcanum found that it collects profile and identity information on Facebook, and URL information on all pages tested, storing all data within the extension's local storage. This is potentially problematic as a vulnerability in the extension could expose this data to malicious websites [20], and the extension may later send this data over network requests.

Academic Projects. COKN Health Info Check [39] is an example extension released as part of an academic research project, which offers fact-checking of health-related web content. Arcanum detected it automatically exfiltrating all sensitive information across all seven pages to an API endpoint at a university, without specifying a privacy policy. While this extension has few users, it highlights the potential privacy risks of mass automated data collection even by researchers, as the data can contain sensitive user information and could be exposed inadvertently if not handled and stored securely.

5 Conclusion and Future Work

In this work, we presented Arcanum, a dynamic taint tracking system for modern Chromium browsers that allows researchers to perform fine-grained analysis of the flow of privacy-sensitive data into and out of extensions. In deploying Arcanum, we discovered extensive privacy risks from thousands of extensions, potentially impacting millions of users. Such risks point to the need for significant changes in the browser extension ecosystem, as well as lessons for web privacy research.

Web Content Matters. Through our study, we found hundreds of extensions that automatically collect the *content* of web pages—in some cases in alarming volume and sensitivity. This highlights the need for tools and systems that account for user data within page content when evaluating web privacy.

Researcher-Driven Annotations Help. Arcanum's fine-grained tracking of specific private user data depends on direct annotation by researchers, for identifying and differentiating various web page components. These annotations are needed not only to understand what flows where, but also to reduce false positives related to non-sensitive content. A future direction in this space is the automatic identification and annotation of web pages, which would allow systems such as Arcanum to scale across sites and discover other privacy problems across the web.

Extension Permissions are Coarse and Opaque. The existing browser extension permissions are coarse, which permits an extension with one stated purpose to risk privacy in other ways. Similarly, in many cases, descriptions of extension behaviors (e.g., privacy policies) do not clearly and accurately articulate the *risk* associated with the behaviors. Moreover, all extensions have the ability to make third-party network requests, adding complications between what an extension can see and what it can do with that information, that may not be apparent to users. Thus, more work is needed for enforcing stricter privacy controls on extensions, as well as driving extensions to deploy accurate privacy policies.

Taint Tracking For Extension Vetting. We strongly encourage the use of systems such as Arcanum and other taint tracking tools as part of the extension vetting process, so that extensions that pose significant privacy risks, even if doing so unintentionally, can be remediated before they impact hundreds of thousands of users.

Limitations and Future work. As discussed in Section 4.1, we do not interact with web pages or extensions during the experiments. However, extensions may exfiltrate sensitive information when specific functions are triggered, and we may not capture such behaviors. Thus, future work can more deeply and extensively investigate extension behavior under interaction and across a broader set of sites. Future work can also combine Arcanum with browser extension fingerprinting methods (e.g., [28, 33]) to detect and notifying users and/or websites about sensitive data exfiltration behaviors by extensions. Furthermore, Arcanum can be optimized more both in terms of performance and functionality, providing an even more effective platform for monitoring extension behaviors.

6 Acknowledgements

This work was in part supported by a gift from Facebook, Inc. The authors also thank David Freeman for insightful discussions, as well as the anonymous reviewers for their valuable feedback.

References

[1] Caniuse. JavaScript operator: Object initializer: Spread properties. https://caniuse.com/?search=Spread%20in%20object, 2023.

[2] Catapult. Web Page Replay. https://chromium.googlesource.com/catapult/+/HEAD/web_page_replay_go/README.md, 2023.

[3] Wentao Chang and Songqing Chen. ExtensionGuard: Towards runtime browser extension information leakage detection. In *IEEE Conference on Communications and Network Security (CNS)*, 2016.

[4] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[5] Chrome. Stable Channel Update for Desktop. https://chromereleases.googleblog.com/2022/11/stable-channel-update-for-desktop_29.html, 2022.

[6] Chrome. API reference. https://developer.chrome.com/docs/extensions/reference/, 2023.

[7] Chrome. Chrome web store sitemap. https://chrome.google.com/webstore/sitemap, 2023.

[8] Chrome. Replace XMLHttpRequest() with global fetch(). https://developer.chrome.com/docs/extensions/migrating/to-service-workers/#replace-xmlhttprequest, 2023.

[9] Chrome. Run time. https://developer.chrome.com/docs/extensions/mv3/content_scripts/#run_time, 2023.

[10] Chrome for Developers. Migrate to Manifest V3. https://developer.chrome.com/docs/extensions/develop/migrate, 2023.

[11] Chrome Web Store has 188k extensions with at least 1.2 billion installs. https://www.ghacks.net/2019/08/04/chrome-web-store-has-188k-extensions-with-at-least-1-2-billion-installs/, 2019.

[12] Chromium. Manifest V3 now available on M88 Beta. https://blog.chromium.org/2020/12/manifest-v3-now-available-on-m88-beta.html, 2020.

[13] Chromium. Will MV2 Chrome extensions stop working on all Chrome browser versions after January 2023. https://groups.google.com/a/chromium.org/g/chromium-extensions/c/xd53CwuOyzk?pli=1, 2022.

[14] Chromium. Legacy JavaScript Implementations. https://chromium.googlesource.com/chromium/src/+/HEAD/extensions/renderer/bindings.md#Legacy-JavaScript-Implementations, 2023.

[15] Catalin Cimpanu. Facebook sues Ukrainian browser extension makers for scraping user data. https://www.zdnet.com/article/facebook-sues-ukrainian-browser-extension-makers-for-scraping-user-data/, 2019.

[16] Catalin Cimpanu. Facebook sues two Chrome extension makers for scraping user data. https://www.zdnet.com/article/facebook-sues-two-chrome-extension-makers-for-scraping-user-data/, 2020.

[17] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[18] MDN Web Docs. Using data attributes. https://https://developer.mozilla.org/en-US/docs/Learn/HTML/Howto/Use_data_attributes, 2023.

[19] MDN Web Docs. Web APIs. https://developer.mozilla.org/en-US/docs/Web/API, 2023.

[20] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[21] Chrome for Developers. Manifest V2 support timeline. https://developer.chrome.com/docs/extensions/develop/migrate/mv2-deprecation-timeline, 2023.

[22] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Security Symposium (USENIX Security)*, 2014.

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

[24] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *ACM SIGSAC conference on Computer & communications security (CCS)*, 2013.

[25] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[26] Meta. How we combat scraping. https://about.fb.com/news/2021/04/how-we-combat-scraping/, 2021.

[27] Meta. Scraping by the numbers. https://about.fb.com/news/2021/05/scraping-by-the-numbers/, 2021.

[28] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *USENIX Security Symposium (USENIX Security)*, 2017.

[29] Selenium. WebDriver. https://www.selenium.dev/documentation/webdriver/, 2023.

[30] Sentry. JavaScript error and performance monitoring. https://sentry.io/for/javascript/, 2023.

[31] Dolière Francis Somé. Empoweb: empowering web applications with browser extensions. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[32] Oleksii Starov and Nick Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *International Conference on World Wide Web (WWW)*, 2017.

[33] Oleksii Starov and Nick Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[34] Chrome Web Store. Avast SafePrice | Comparison, deals, coupons. https://chrome.google.com/webstore/detail/avast-safeprice-compariso/eofcbnmajmjmplflapaojjnihcjkigck, 2023.

[35] Chrome Web Store. Avira Browser Safety. https://chrome.google.com/webstore/detail/avira-browser-safety/flliilndjeohchalpbbcdekjklbdgfkk, 2023.

[36] Chrome Web Store. Avira Safe Shopping. https://chrome.google.com/webstore/detail/avira-safe-shopping/ccbpbkebodcjkknkfkpmfeciinhidaeh, 2023.

[37] Chrome Web Store. Bookmark more. https://chrome.google.com/webstore/detail/bookmark-more/jdleicahfbehiikjcaocollfhbnigplo, 2023.

[38] Chrome Web Store. Capital One Shopping: Add to Chrome for Free. https://chrome.google.com/webstore/detail/capital-one-shopping-add/nenlahapcbofgnanklpelkaejcehkggg, 2023.

[39] Chrome Web Store. COKN Health Info Check. https://chrome.google.com/webstore/detail/cokn-health-info-check/blcdkmjcpgjojjffbdkckaiondfpoglh, 2023.

[40] Chrome Web Store. DetectGPT - Detect Chat GPT Content. https://chrome.google.com/webstore/detail/detectgpt-detect-chat-gpt/oadkgbgppkhoaaoepjbcnjejmkknaobg, 2023.

[41] Chrome Web Store. Eco-Index by Changing Room. https://chrome.google.com/webstore/detail/eco-index-by-changing-roo/pjmfidajplecneclhdghcgdefnmhhlca, 2023.

[42] Chrome Web Store. Email Extractor. https://chrome.google.com/webstore/detail/email-extractor/jdianbbpnakhcmfkcckaboohfgnngfcc, 2023.

[43] Chrome Web Store. Email Finder-Kendo Sourcing Ninja. https://chrome.google.com/webstore/detail/email-finder-kendo-sourci/kecadfolelkekbfmmfoifpfalfedeljo, 2023.

[44] Chrome Web Store. Fiction Reader. https://chrome.google.com/webstore/detail/%E5%B0%8F%E8%AF%B4%E9%98%85%E8%AF%BB%E5%8A%A9%E6%89%8B/dknlfmhongfkfakmhhnmgfgnhhcbmldm, 2023.

[45] Chrome Web Store. GoRateUp. https://chrome.google.com/webstore/detail/gorateup/opmmfaampmbhbohaaamhfpennnefnkfn, 2023.

[46] Chrome Web Store. Hola VPN - The Website Unblocker. https://chrome.google.com/webstore/detail/hola-vpn-the-website-unbl/gkojfkhlekighikafcpjkiklfbnlmeio, 2023.

[47] Chrome Web Store. Honey: Automatic Coupons & Rewards. https://chrome.google.com/webstore/detail/honey-automatic-coupons-r/bmnlcjabgnpnenekpadlanbbkooimhnj, 2023.

[48] Chrome Web Store. Investor Intel. https://chrome.google.com/webstore/detail/investor-intel/onfmefagepefndfhefodadmpodcdcneh, 2023.

[49] Chrome Web Store. Likewise. https://chrome.google.com/webstore/detail/likewise/bahcihkpdjlbndandplnfmejnalndgjo, 2023.

[50] Chrome Web Store. Mr. E - Find B2B contacts universally. https://chrome.google.com/webstore/detail/mr-e-find-b2b-contacts-un/haphbbhhknaonfloinidkcmadhfjoghc, 2023.

[51] Chrome Web Store. NordVPN - VPN Proxy for Privacy and Security. https://chrome.google.com/webstore/detail/nordvpn-vpn-proxy-for-pri/fjoaledfpmneenckfbpdfhkmimnjocfa, 2023.

[52] Chrome Web Store. Online Security. https://chrome.google.com/webstore/detail/online-security/llbcnfanfmjhpedaedhbcnpgeepdnnok, 2023.

[53] Chrome Web Store. PowerAdSpy - Ad Intelligence. https://chrome.google.com/webstore/detail/poweradspy-ad-intelligenc/nkecaphdplhfmmbkcfnknejeonfnifbn, 2023.

[54] Chrome Web Store. QuillBot: AI Grammar and Writing Tool. https://chrome.google.com/webstore/detail/quillbot-ai-grammar-and-w/iidnbdjijdkbmajdffnidomddglmieko, 2023.

[55] Chrome Web Store. SHADE: Stylishly Sustainable. https://chrome.google.com/webstore/detail/shade-stylishly-sustainab/mdfgkcdjgpgoeclhefnjgmollcckpedk, 2023.

[56] Chrome Web Store. Sidex Price Scanner. https://chrome.google.com/webstore/detail/%D1%81%D0%B0%D0%B9%D0%B4%D0%B5%D0%BA%D1%81-%D1%81%D0%BA%D0%B0%D0%BD%D0%B5%D1%80-%D1%86%D0%B5%D0%BD/aamfmnhcipnbjjnbfmaoooiohikifefk, 2023.

[57] Chrome Web Store. Touch VPN - Secure and unlimited VPN proxy. https://chromewebstore.google.co

m/detail/touch-vpn-secure-and-unli/bihmplhobchoageeokmgbdihknkjbknd, 2023.

[58] Top browsers market share. https://www.similarweb.com/browsers/, 2023.

[59] V8. A Read-only space in V8. https://docs.google.com/document/d/1UxALqYAnmUnajDmswvizD7c5_pqQ1ks5wATry-nNBLA/edit, 2018.

[60] V8. Isolate Independent HeapObjects. https://docs.google.com/document/d/1awXj2nt4xDKoAO1iVDUDg51oGOTkgBR0WCcpkUldrUo/edit, 2018.

[61] V8. CodeStubAssembler builtins. https://v8.dev/docs/csa-builtins, 2020.

[62] V8. Pointer Compression in V8. https://v8.dev/blog/pointer-compression, 2020.

[63] V8. V8 Torque builtins. https://v8.dev/docs/torque-builtins, 2023.

[64] V8. Launching Ignition and TurboFan. https://v8.dev/blog/launching-ignition-and-turbofan, May. 2017.

[65] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2007.

[66] Michael Weissbacher. These Chrome extensions spy on 8 million users. https://mweissbacher.com/2016/03/31/these-chrome-extensions-spy-on-8-million-users/, 2016.

[67] Michael Weissbacher, Enrico Mariconti, Guillermo Suarez-Tangil, Gianluca Stringhini, William Robertson, and Engin Kirda. Ex-ray: Detection of history-leaking browser extensions. In *Annual Computer Security Applications Conference (ACSAC)*, 2017.

[68] Mengfei Xie, Jianming Fu, Jia He, Chenke Luo, and Guojun Peng. JTaint: Finding Privacy-leakage in Chrome Extensions. In *Information Security and Privacy: Australasian Conference (ACISP)*, 2020.

[69] Xvfb virtual framebuffer X server for X Version 11. https://manpages.ubuntu.com/manpages/trusty/man1/Xvfb.1.html, 2023.

A Artifact Appendix

A.1 Abstract

In this work, we develop Arcanum, a dynamic taint tracking system for modern Chrome extensions designed to monitor the flow of sensitive user content from web pages. Arcanum defines a diverse set of taint sources ranging from meta-data, to content DOM elements, location information, history data, and cookies. From these sources, Arcanum is able to track data flow to a variety of exit taint sinks, including all forms of web requests and storage APIs. A key feature of Arcanum is allowing researchers to instrument specific web page elements as tainted at runtime via JS DOM annotations. We deploy Arcanum to test all functional extensions currently in the Chrome Web Store for the automated exfiltration of user data across seven sensitive websites: Amazon, Facebook, Gmail, Instagram, LinkedIn, Outlook, and PayPal. We observe significant privacy risks across thousands of extensions, including hundreds of extensions automatically extracting user content from within web pages.

The Arcanum prototype is built on Chromium Browser version 108.0.5359.71. We open-source all Chromium patches of the Arcanum implementation, allowing users to build and adapt Arcanum from the Chromium source code. In the artifacts, we provide custom Chrome extensions for testing Arcanum's functionality, as well as representative real-world extensions that were evaluated and flagged by Arcanum.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

When evaluating extension behavior on a web page using Arcanum, we strongly recommend users to record that web page and replay it across the extensions evaluated using WprGo. By only visiting the pages once and replaying the captured response, we can minimize additional load on a site regardless of the number of extensions evaluated. We provide web page recordings for all experimental target pages in this artifact. We also note that Arcanum itself does not require replay and can be run on live web pages.

A.2.2 How to access

This artifact, including 1) all Chromium patches of the Arcanum implementation, 2) sample extensions, 3) JavaScript files for annotating specific DOM elements on web pages, 4) web page recordings, and 5) Python test case scripts for each sample extension, is hosted on GitHub and can be accessed via: `https://github.com/BEESLab/Arcanum/releases/tag/1.0`. Additionally, we provide two Docker images on Docker Hub that have the necessary dependencies for 1) building Arcanum from the Chromium source code, and 2) running Arcanum to test sample extensions. They can be pulled via "`docker pull xqgtiti/arcanum_build:latest`", and "`docker pull xqgtiti/arcanum_run:latest`", respectively.

A.2.3 Hardware dependencies

Our experiments can be conducted either directly on a single physical host machine or alternatively in a VM-based lab environment on a host machine.

A x86_64 (amd64) machine with at least 8 GiB RAM, 4 cores/8 threads CPU, and at least 100 GiB of free disk space is required. More than 16 GiB RAM is highly recommended. For reference, all our experiments were conducted on a physical machine with 512 GiB RAM, 32 cores/128 threads CPU, running Ubuntu 20.04.6 LTS (Kernel Linux 5.4.0-173-generic). The provided test cases were also evaluated on another Linux server with 8 CPUs and 16 GiB RAM.

A.2.4 Software dependencies

Git is required for fetching the Chromium source code 108.0.5359.71 (Linux) and depot tools.

To build Arcanum, we provide a Docker image (Ubuntu 20.04) that includes all necessary dependencies for building a version of Chromium patched with the Arcanum implementation. Alternatively, users can follow the official instruction to build your own Docker container.

To run Arcanum, we provide a Docker image (Ubuntu 18.04) that includes all necessary dependencies. We strongly recommend using this pre-configured image, as our test case scripts partly rely on its settings (such as software executable paths). If you choose to build the environment manually, the following software dependencies are required:

- Go 1.19.12 Linux
- WprGo v0.0.0-20230901234838-f16ca3c78e46
- Python 3.8.0 Linux (with Selenium 4, pyvirtualdisplay)
- ChromeDriver 108.0.5359.71
- Xvfb
- Chromium Dependencies

While we have not verified compatibility with versions other than those listed above, we believe that our artifacts will work with Ubuntu 18.04, 20.04, and 22.04, Python 3.8+ (Selenium 4), and any versions of Xvfb and WprGo.

A.2.5 Benchmarks

To benchmark the performance of Arcanum, we provide two types of sample extensions in this artifact as the dataset:

Custom Extensions. We provide sample extensions implemented by ourselves to demonstrate how extensions can be tested using Arcanum, on seven websites that were experimented with in our paper (Amazon, Facebook, Gmail, Instagram, LinkedIn, Outlook, and PayPal). For each site, we provide one Manifest Version 2 (MV2) extension and one

Manifest Version 3 (MV3) extension. In addition to these extensions, we also provide custom extensions to guide users in testing the taint tracking process of Arcanum, including testing different taint sources, sinks, and propagation cases.

Real-world Extensions. We provide representative extensions from the Chrome Web Store that have been tested and flagged by Arcanum. Specifically, we include all extensions discussed as case studies of Section 4.10, as well as those listed in Table 7 of Section 4.5 (i.e., the flagged extensions with the most users in each web content category). The extension IDs are listed in Table 9:

Extension ID	Paper Section
aamfmnhcipnbjjnbfmaoooiohikifefk	Case Study, Table 7
haphbbhhknaonfloinidkcmadhfjoghc	Case Study, Table 7
jdianbbpnakhcmfkcckaboohfgnngfcc	Case Study, Table 7
oadkgbgppkhoaaoepjbcnjejmkknaobg	Case Study, Table 7
blcdkmjcpgjojjffbdkckaiondfpoglh	Case Study
kecadfolelkekbfmmfoifpfalfedeljo	Table 7
nkecaphdplhfmmbkcfnknejeonfnifbn	Table 7
bahcihkpdjlbndandplnfmejnalndgjo	Table 7
pjmfidajplecneclhdghcgdefnmhhlca	Table 7
mdfgkcdjgpgoeclhefnjgmollcckpedk	Table 7

Table 9: Real-world extension IDs.

Web Page Recording Files. Since popular sites periodically mutate to combat automation/scraping, we provide recording files for each target page of the seven websites (as listed in Table 3) for consistent evaluation. JS scripts for annotating privacy-sensitive DOM elements on each web page are also provided. All sample extensions mentioned above are tested by Arcanum using these recording files in this artifact.

A.3 Set-up

A.3.1 Installation

This section describes how to set up a build environment for Chromium and build a version of Chromium with the patches of the Arcanum implementation. The set-up is mostly based on the official Chromium build instructions on Linux.

1. On the host machine, clone the Chromium depot tools to a specific directory (e.g., `$HOME`) and add their path to the PATH environment variable.

```
git clone https://chromium.googlesource.
    ↪ com/chromium/tools/depot_tools.git
export PATH="${HOME}/depot_tools:$PATH"
```

2. Get the Chromium source code (this may take a while depending on your network connection).

```
mkdir ${HOME}/chromium && cd ${HOME}/
    ↪ chromium/
fetch --nohooks chromium
```

3. In `src/`, check out the branch for Chromium version 108.0.5359.71. You could also refer to the official instructions on working with Chromium release branches.

```
cd src
gclient sync --with_branch_heads --
    ↪ with_tags
git fetch
git checkout tags/108.0.5359.71
gclient sync --with_branch_heads --
    ↪ with_tags
```

4. Prepare the Docker container for building. Pull the provided Docker image for building, then launch a Docker container from this image. Make sure to mount the host directory containing the Chromium source code and the depot_tools into the container:

```
docker pull xqgtiti/arcanum_build:latest
docker run -it --mount src=${HOME},target
    ↪ ="/mnt/build/",type=bind --name=
    ↪ build xqgtiti/arcanum_build:latest
```

5. Prepare build in the Docker container's interactive shell. Add the path of Chromium depot tools to the PATH environment variable. The command "`gn args ...`" automatically opens a file (`args.gn`) in the default text editor. Replace the contents of this file with the contents of the file "`~/build/args.gn`" from the artifact GitHub repository.

```
export PATH="/mnt/build/depot_tools:$PATH"
cd /mnt/build/chromium/src/
gn args out/Default
```

6. After updating the contents of the `args.gn` file, run the above command again to finalize the build preparations.

```
gn args out/Default
```

7. Build an unmodified Chrome and its Linux installer (this may take a while depending on the host machine's performance).

```
ninja -C out/Default chrome
ninja -C out/Default "chrome/installer/
    ↪ linux:unstable_deb"
```

8. Build a version of Chrome patched with the Arcanum implementation (located in "`~/patches/`" in the artifact GitHub repository).

```
cd /mnt/build/chromium/src/
git apply ~/patches/chromium.patch
cd /mnt/build/chromium/src/v8/
git apply ~/patches/v8.patch

cd /mnt/build/chromium/src/
gn args out/Arcanum
cp out/Default/gn.args out/Arcanum/
gn args out/Arcanum
ninja -C out/Arcanum chrome
```

9. Build a Linux installer for Arcanum, you can then find the `.deb` file in "`../out/Arcanum/`".

```
ninja -C out/Arcanum "chrome/installer/
    ↪ linux:unstable_deb"
cd /mnt/build/chromium/src/out/Arcanum/
ls chromium-browser-unstable_108
    ↪ .0.5359.71-1_amd64.deb
```

A.3.2 Basic Test

Pull the provided Docker image for running Arcanum, and then launch a Docker container from this image. Note that the "`--privileged`" flag is required. You can also mount any directory that is convenient for transferring files from the host machine.

```
docker pull xqgtiti/arcanum_run:latest
docker run -it --privileged --name=run
    ↪ xqgtiti/arcanum_run:latest
```

Copy the Arcanum installer file (i.e., the `.deb` file) to "`/root/Arcanum/`" in the Docker container and decompress it. Note that we use this path in the test case code as the Arcanum executable path. Please modify the variable in the code if you place the installer elsewhere.

```
cd /root/Arcanum/
ar x chromium-browser-unstable_108
    ↪ .0.5359.71-1_amd64.deb
tar -vxf control.tar && tar -vxf data.tar
```

Run the basic test case `Basic_Test.py` from the artifact GitHub repository in the interactive shell of the container, using the pre-configured Python 3.8. The basic test case uses Selenium to launch Arcanum (a modified Chromium) with an empty extension pre-installed and navigates to a web page.

```
python3.8 ~/Test_Cases/Basic_Test.py
```

If Arcanum runs normally, you should see "`Basic Test: Success.`" in the output.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The Functionality of Arcanum.

- Arcanum operates on the modern browser architecture, as proven by the implementation patches for Chromium 108.
- Arcanum supports both MV2 and MV3 extensions. This is proven by experiment E1, where we test custom extensions for both MV2 and MV3.
- Arcanum is able to track user sensitive data from within web pages and support taint propagation across a comprehensive set of browser, web, and JavaScript APIs used by extensions. This is proven by both experiments E1 and E2. In E1, we test custom extensions that trigger different types of taint sources (including `chrome.history`, `document.location`, etc.), taint sinks (including `Fetch`, `XMLHttpRequest`, etc.), and various propagation processes (including binary data buffer propagation, Chrome message passing APIs, etc.), as described in Section 3. In E2, we test real-world extensions covering more propagation cases, such as the storage taint sink, literal creation for compound JS types, JS Promise, etc.
- Arcanum produces detailed taint propagation logs. This is proven by both experiments E1 and E2, where the test case results are determined by whether the expected content appears in the taint logs.

(C2): Arcanum Can Track Specific Web Page Content. Arcanum allows researchers to instrument specific web page elements as tainted at runtime via JS DOM annotations. This is proven by both experiments E1 and E2, where we annotate specific sensitive DOM elements on seven target pages and test the extensions with these annotations.

(C3): Our Experiments are Reproducible. In deploying Arcanum, we are able to discover automated exfiltration of user data by real-world extensions across seven popular websites. This is proven by experiment E2 described in Section 4.5 and Section 4.10. Specifically, we provide all extensions discussed as case studies in Section 4.10 and those whose IDs are listed in Table 7 of Section 4.5. Our test cases evaluate whether Arcanum can successfully flag each sample extension, identifying whether the extension collects specific tainted page content on a target page and propagates the information to a taint sink.

A.4.2 Experiments

(E1): [Test Custom Extensions] [1 human-hour]
Preparation: Use the same Docker container from the Basic Test that has the Arcanum executable file placed in "`/root/Arcanum/`". Download all custom extensions (in "`~/Sample_Extensions/Custom/`") from the artifact GitHub repository and copy the extensions to "`/root/extensions/custom/`" in the container.

```
mkdir -p /root/extensions/custom/
cp -r ~/Sample_Extensions/Custom/* /root
    ↪ /extensions/custom/
```

Download all recordings and JS scripts for DOM element annotations from the artifact GitHub repository and place them in the "`/root/`" directory in the container:

```
mkdir -p /root/recordings/
cp -r ~/recordings/* /root/recordings/
mkdir -p /root/annotations/
cp -r ~/annotations/* /root/annotations/
```

Execution: We have prepared a test case for each custom extension. Run these test cases in the container shell using the pre-configured Python 3.8:

```
python3.8 ~/Test_Cases/Custom_Test.py
```

Each test case launches Arcanum with the corresponding web recording and DOM element annotations (or without annotations when testing non-web content taint sources), and checks whether we successfully obtain the expected content in the taint logs, demonstrating the correct taint tracking of Arcanum. You can test all custom extensions together or test a specific extension by simply commenting out other test cases in `Custom_Test.py`, such as:

```
if __name__ == "__main__":
    Amazon_Extension_MV2_Test()
    # Amazon_Extension_MV3_Test()
    # Facebook_Extension_MV2_Test()
    ...
    # Source_document_password_Test()
    # Source_document_location_Test()
    ...
```

Results: For each extension being tested, you should see "Custom Extension ${Name}: Success." in the test case output, demonstrating the correct taint tracking of Arcanum. You can refer to the test case code to see the expected content in the taint logs for each extension.

(E2): [Test Real-world Extensions] [1 human-hour]

Preparation: Use the same Docker container from the Basic Test that has the Arcanum executable file placed in "`/root/Arcanum/`". Download all real-world extensions (in "`~/Sample_Extensions/Realworld/`") from the artifact GitHub repository and copy the extensions to "`/root/extensions/realworld/`" in the container.

```
mkdir -p /root/extensions/realworld/
cp -r ~/Sample_Extensions/Realworld/*
    ↪ /root/extensions/realworld/
```

Execution: We have prepared a test case for each real-world extension. Run these test cases in the container shell using the pre-configured Python 3.8:

```
python3.8 ~/Test_Cases/Realworld_Test.py
```

Each test case launches Arcanum with the corresponding web recording and DOM element annotations, and checks whether we successfully obtain the expected content in the taint logs, aligning with the experiment results described in Sections 4.5 and Section 4.10. You can test all real-world extensions together or test a specific extension by simply commenting out other test cases in `Realworld_Test.py`, such as:

```
if __name__ == "__main__":
    amfmnhcipnbjjnbfmaoooiohikifefk()
    # haphbbhhknaonfloinidkcmadhfjoghc()
    # dianbbpnakhcmfkcckaboohfgnngfcc()
    ...
```

Results: For each extension being tested, you should see "Real-world Extension ${ID}: Success." in the test case output. You can refer to the test case code to see the expected content in the taint logs for each extension. We also release all taint logs (i.e., analysis results generated by Arcanum) for each real-world extension obtained from the experiments conducted in our paper. Please check these logs located in ∼`/Taint_Logs/` in the artifact GitHub repository.

A.5 Notes on Reusability

- Arcanum's taint source logs, propagation logs, and the storage sink logs are located in "`/ram/analysis/v8logs/`". All other taint sink logs are in the specified user data directory of Chromium.
- When testing Arcanum with Docker, ensure to allocate sufficient CPU resources (4 logical processors or more), especially when running multiple containers in parallel (e.g., using "`--cpus=4 --cpuset-cpus=0-3`"). Use "`--cpuset-cpus`" to specify CPUs in scenarios where preemption may occur.
- As described in Section 3.4, we introduce a forced delay in Arcanum to ensure that a target web page will fully load before an extension injects its content script. We configure this delay as a Chrome switch "`--custom-script-idle-timeout-ms`" and "`--custom-delay-for-animation-ms`". Users can set a specific delay when recording and replaying different web pages according to their page loading times. Please refer to the provided test cases for examples of its usage. The test cases were evaluated on a Linux server with 8 CPUs and 16 GiB of RAM. If you are testing with fewer CPU resources, please consider increasing the value of the two switches mentioned above in the test case scripts.
- Please see the README file in our GitHub repository for future updates.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2024/`.