

Article

The Blind Spot of Extension Security: WebAssembly–JavaScript Collaborative Attacks on Chrome

Yeongmin Moon ¹, Minhyuk Hong ² and Jeman Park ^{2,*}

¹ Department of Computer Engineering, Kyung Hee University, Yongin-si 17104, Republic of Korea; eoeung113@khu.ac.kr

² Department of Software Convergence, Kyung Hee University, Yongin-si 17104, Republic of Korea; minster7650@khu.ac.kr

* Correspondence: jeman@khu.ac.kr

Abstract

Chrome extensions are increasingly exploited as an attack surface, yet existing static malware detectors share a critical blind spot: they analyze JavaScript but cannot inspect WebAssembly (Wasm) or reason across the Wasm–JS boundary. We exploit this gap by embedding malicious logic in Wasm modules while confining JavaScript to minimal glue code, rendering the core of each attack invisible to static analysis. Grounded in this collaborative architecture, we implement eight proof-of-concept attack scenarios across six categories—adware, malicious file delivery, forced redirection, keylogging, credential theft, and ransomware—as functioning Manifest V3 extensions. Evaluated against four representative static detectors, none achieves genuine detection: three register the samples as benign or raise no alert, and the fourth flags every sample only through systematic false positives on generated glue code. An analysis of 165,314 live extensions further shows that every permission our attacks require is already in widespread legitimate use, so such extensions would not be distinguishable from benign ones by permission-based screening alone.

Keywords: browser extension; WebAssembly; evasion attack

1. Introduction

Chrome extensions enhance browser functionality and usability, yet they simultaneously introduce an attack surface that adversaries increasingly exploit [1–3]. Attackers distribute malicious extensions that exfiltrate users’ personal data [4–7] and silently redirect victims to phishing sites [8]. With an estimated user base exceeding 900 million [1,9], the extension ecosystem enables supply-chain attacks at unprecedented scale.

The threat has grown more acute in recent years. A coordinated supply-chain campaign compromised dozens of legitimate extensions by hijacking developer accounts, injecting data-exfiltration logic that silently reached millions of users before detection [10,11]. Separately, extensions leveraging AI-driven automation have begun accessing sensitive browser state without explicit user interaction [12]. These incidents share a structural thread: malicious logic concealed within what appears to be ordinary extension code. Our work demonstrates that this concealment problem is deeper than previously recognized: attackers need not hijack legitimate extensions or compromise developer accounts—a benign-looking extension suffices when the analysis tools themselves cannot see the execution layer in which the malicious logic resides.



Academic Editor: Mariusz Sepczuk

Received: 11 June 2026

Revised: 7 July 2026

Accepted: 8 July 2026

Published: 11 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

The threat has attracted substantial research attention, yielding a range of static and dynamic detection techniques. Early approaches extract structural features from JavaScript code and manifest files and apply machine learning classifiers [13,14]; more recent work introduces static data-flow analysis over extension JavaScript [15] and supervised classification at Chrome Web Store scale [16], achieving considerable detection accuracy. These detectors treat the extension as a JS artifact: they parse abstract syntax trees, extract token n-grams, or monitor DOM events [17,18]—all operations that function exclusively within the JS runtime and produce no signal when malicious logic is absent from JS entirely. However, these advances share a common assumption: that malicious logic resides in JS. As a result, existing detectors largely overlook WebAssembly (Wasm), a binary instruction format that has become an integral part of modern web standards.

Originally designed to deliver near-native execution performance in the browser, Wasm's compact binary format is inherently more difficult to inspect than JS source code. We observe that this analysis gap creates a practical evasion opportunity: an attacker can implement malicious functionality in Wasm while keeping only benign-looking glue code in JS, exploiting the cross-language boundary that current detectors do not traverse. In practice, this means that target URLs, encryption keys, file payloads, and exfiltration endpoints can all reside inside a Wasm binary—opaque to any tool that does not disassemble and semantically interpret Wasm bytecode. The JS component of such an extension need only contain innocuous glue code to instantiate the module and relay results to browser APIs, exposing no pattern that static analyzers can flag. Prior work has shown that translating malicious JS logic into Wasm dramatically reduces detection rates by JS-oriented analyzers [19], yet no study has systematically examined how this property can be weaponized within the Chrome extension ecosystem.

In this paper, we present a systematic study of Wasm-assisted malicious Chrome extensions. We first define an attack model grounded in the Wasm-JS collaborative architecture, enumerating attack types and the capability chains required to realize each. We then implement a proof-of-concept (PoC) extension covering six attack categories—including adware, keylogging, credential theft, and ransomware—and evaluate it against four representative static detectors spanning JS-based analysis (Cujo [13], JStap [14]), direct Wasm-binary analysis (WasmGuard [20]), and the deployed commercial engine ecosystem (VirusTotal [21]). All PoCs were executed in a local environment and were never submitted to the Chrome Web Store, in accordance with responsible disclosure principles.

Our evaluation reveals a consistent and critical blind spot: JStap and WasmGuard classified every malicious PoC as benign, and VirusTotal's 61–62 conventional antivirus engines likewise raised no alerts. Cujo flagged all samples, but further analysis showed that it does so indiscriminately—any extension containing Wasm-related glue code is treated as malicious regardless of actual intent, yielding an impractical false-positive rate rather than genuine detection capability. These results demonstrate that the static detection surface gating the extension ecosystem—research analyzers and the deployed commercial engine set alike—offers no reliable defense against Wasm-assisted extension attacks. This failure is not incidental but structural: each detector—whether JS-based or Wasm-aware—reasons about a single execution layer in isolation and cannot trace the semantics that cross the Wasm-JS boundary.

The implications extend beyond the extensions we tested. Because Wasm modules and their JS glue code are both explicitly permitted under Manifest V3 (MV3)—Google's current, stricter extension model—the attack surface is not an artifact of legacy permissiveness but an inherent property of the platform. An adversary distributing a Wasm-assisted malicious extension today is genuinely detected by none of the static detectors we evaluate, and could expose tens of thousands of users—or more, given the heavy-tailed popularity of extensions

holding the required permissions—before any behavioral signal emerges. Dynamic analysis can in principle observe these runtime effects, but applying it to continuous Chrome Web Store screening would be constrained by cost, and attacks gated behind specific victim conditions further limit its reliability—points we revisit in Section 5, where we also quantify the ecosystem-scale exposure in concrete terms.

The main contributions of this paper are as follows:

- We identify and characterize the Wasm–JS cross-language architecture as a practical evasion vector in the Chrome extension ecosystem.
- We design an attack model that characterizes six attack categories—realized as eight concrete scenarios—and maps the capability chains required to implement each.
- We implement and evaluate a PoC malicious extension suite, showing that no tested static detector—JS-based or Wasm-aware—achieves genuine detection, with Cujo’s flags shown to be indiscriminate false positives.
- We quantify the real-world feasibility of the attack model through an analysis of 165,314 live Chrome Web Store extensions, showing that every permission it requires is already in widespread legitimate use.
- We discuss why single-layer analysis fails against this threat model and outline cross-language directions for Wasm-aware extension security.

2. Background and Motivation

2.1. Chrome Extension Architecture

A Chrome extension is composed of a manifest file (`manifest.json`), HTML/CSS for the user interface, and JS for event handling and browser interaction. The manifest file declares the extension’s metadata, required permissions, and entry points, including background scripts and content scripts. Background scripts run persistently in the browser background and have access to the Chrome extension APIs, while content scripts are injected into web pages and interact with the DOM. This architecture grants extensions broad capabilities—such as intercepting network requests, reading page content, and managing downloads—making them an attractive target for abuse.

Chrome extensions have historically operated under Manifest Version 2 (MV2), which permitted highly permissive configurations, including remotely hosted code execution and unrestricted background processing. These properties enabled attacks such as covert cryptocurrency mining: one documented case involved the *Archive Poster* extension, which used WebSocket connections to fetch malicious Wasm modules and perform CPU-intensive mining across more than 105,000 users’ browsers [22,23]. Google has since deprecated MV2 in favor of Manifest Version 3 (MV3), which enforces stricter permission scoping and prohibits remotely hosted scripts [24]. While MV3 raises the barrier for certain attack classes, it does not eliminate the threat: the combination of legitimate-looking JS glue code and binary Wasm modules—both permitted under MV3—remains a viable and underexplored attack surface.

2.2. WebAssembly in the Browser

WebAssembly (Wasm) is a binary instruction format designed to execute at near-native speed inside web browsers [25]. It is produced by compiling high-level languages such as C, C++, or Rust, and is supported by all major browsers including Chrome, Firefox, Safari, and Edge [26]. Wasm modules interact with the surrounding JS environment through a well-defined import/export interface: JS code instantiates a Wasm module and calls its exported functions, while the module may invoke JS-defined functions as imports. This interoperability is mediated by *glue code*—a JS wrapper that manages memory, marshals data types, and bridges the two runtimes. Figure 1 illustrates this structure with a representative benign example: an image filter implemented in C/C++ operates on pixel

data in Wasm linear memory, and JS glue code copies the result into a browser canvas via `putImageData()`.

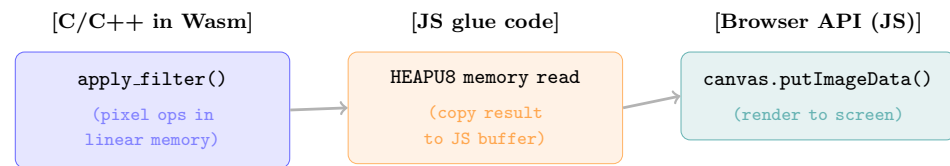


Figure 1. A Representative Wasm–JS Interaction. An image filter implemented in C/C++ operates on pixel data in Wasm linear memory, and JS glue code copies the result into a browser canvas via `putImageData()`.

In practice, glue code is typically generated automatically by the Emscripten toolchain [27], which compiles C/C++ source into a `.wasm` binary paired with a JS file that handles module instantiation and memory management. This generated JS exposes a set of typed array views over the Wasm linear memory (e.g., `HEAPU8`, `HEAPF64`) and provides virtual filesystem abstractions—MEMFS for session-scoped in-memory storage and IDBFS for persistent IndexedDB-backed storage—that allow Wasm modules to perform file I/O without any browser-native file API call appearing in JS source. Critically, the JS glue code produced by this toolchain is structurally identical regardless of what the Wasm module computes: whether the module performs image filtering or constructs a malicious redirect URL, the surrounding JS contains only memory access patterns and API relay calls, with no application-level semantics visible to a static analyzer. From an attacker’s perspective, this means that file payloads, network endpoints, and cryptographic material can be embedded directly in the compiled Wasm binary, while the glue code that surfaces in JS contains nothing semantically suspicious—a property compounded by the fact that Wasm bytecode itself is opaque without dedicated disassembly tooling, unlike JS which is human-readable and amenable to pattern-based static analysis. Indeed, prior work on the binary security of Wasm has shown that its low-level memory model and compact bytecode make Wasm binaries substantially harder to inspect and reason about than equivalent native or source-level code [28].

An early large-scale study of Wasm deployment found that the majority of Wasm usage on the web was associated with malicious activities such as cryptomining and code obfuscation, rather than legitimate performance optimization [23]. Subsequent measurement of real-world Wasm binaries confirmed that Wasm has since become a routine component of the modern web platform, broadening the attack surface well beyond these early cases [29]. This suggests that adversaries have long recognized Wasm’s evasion potential—yet its implications for the Chrome extension ecosystem remain largely unexamined.

2.3. Limitations of Existing Detection Approaches

Existing malicious extension detectors predominantly operate on JS artifacts. Static approaches extract features from JS code and manifest files [13,14]; dynamic approaches monitor runtime JS behavior [17,18]. Dynamic analyzers—such as those that instrument content scripts at runtime or intercept Chrome API calls—share the same fundamental constraint: they observe only the JS execution context. JS-level dynamic instrumentation does not resolve this constraint. Behavioral signals produced entirely within the Wasm runtime—internal control flow, intermediate computation over sensitive data, and the semantic interpretation of values crossing the JS–Wasm boundary—do not surface as observable JS API calls and therefore remain invisible to any JS-level hook, as demonstrated concretely by the attack scenarios in Section 3.4. This limitation is compounded by the absence of dynamic analysis tools that jointly model the Wasm–JS boundary within the browser extension environment: existing dynamic analyzers either target general web

pages without accounting for Chrome-specific APIs, or instrument only the JS execution context without visibility into the Wasm runtime. We therefore center our evaluation on representative static detectors spanning JS-based analysis, direct Wasm-binary analysis, and the deployed commercial engine ecosystem. While platform review processes such as Chrome Web Store vetting incorporate dynamic and behavioral signals, prior work has shown that malicious extensions still pass such review [6,30]; we revisit why cross-language attacks remain difficult to catch even under dynamic analysis in Section 5.3.

The severity of this gap has been partially demonstrated in prior work. Wobfuscator [19] showed that converting malicious JS logic into Wasm dramatically reduces detection rates by JS-based analyzers, as the detectors lack any mechanism to interpret binary Wasm semantics. WasmGuard [20], the only known detector that directly analyzes Wasm binaries, is a general-purpose malicious Wasm detector that relies solely on static binary analysis; as a result, it cannot account for the collaborative Wasm–JS execution model that characterizes extension-based attacks. The closest effort toward cross-language detection, JWBinder [31], reconstructs JS–Wasm data flow for general web malware, but it targets web pages rather than the extension environment. We refer to this unmonitored region between the two runtimes as the *cross-language gap*: the set of data flows and control transfers that originate in Wasm, cross the import/export boundary into JS, and ultimately invoke browser APIs—a chain that no existing static detector reconstructs end-to-end.

Taken together, these limitations reveal a practical evasion opportunity: an attacker can encode malicious functionality in Wasm while exposing only innocuous glue code in JS, rendering the extension invisible to existing JS-based detectors. This observation motivates the attack model and PoC implementation presented in the following sections.

3. Attack Model and PoC Implementation

This section describes the design and implementation of the proposed attack model. We first define the research questions and scope, then present the overall attack flow, enumerate the capability primitives available to a malicious extension, and detail the eight attack scenarios realized in our PoC.

3.1. Research Questions and Scope

This study addresses two research questions:

- **RQ1:** What attack types are feasible when malicious logic is implemented in WebAssembly within a Chrome extension?
- **RQ2:** Can existing static malware detectors reliably identify Chrome extensions that conceal malicious behavior in WebAssembly?

We operate under the following assumptions. First, malicious logic resides primarily in Wasm, with JS limited to functions that Wasm cannot perform natively: module invocation, network communication, event handling, and message passing. Second, Wasm modules are compiled from C/C++ using the Emscripten toolchain [27]. Third, the attacker distributes a seemingly legitimate extension and does not exploit known browser vulnerabilities; all actions remain within the permissions declared in the manifest. Fourth, the attack surface is confined to the browser environment of the user who has installed the extension.

We further specify the attacker’s capabilities and deployment. The attacker is an extension developer—either publishing a new extension or having compromised a legitimate developer’s account—who distributes through the standard Chrome Web Store channel and controls a server used for proxy routing and exfiltration. Distribution follows ordinary vectors: store search, direct links, or social engineering in which the extension is presented as a useful utility.

All PoC extensions were executed in a local research environment and were not submitted to the Chrome Web Store, in accordance with responsible disclosure principles.

3.2. Overview of Attack Flow

Figure 2 illustrates the three architectural patterns that determine whether a malicious extension is detectable by existing tools. In the *JS-only* architecture, all malicious logic resides in JS and is visible to JS-based static analyzers. In the *Wasm-only* architecture, malicious logic is entirely contained within a Wasm binary; a dedicated Wasm analyzer can detect characteristic patterns such as high arithmetic density or anomalous memory usage. In the *Wasm–JS collaborative* architecture—the focus of this study—malicious logic is distributed across both runtimes. Wasm implements the core malicious computation (e.g., encryption routines, exfiltration target URLs, redirect destinations), while JS manages control flow and browser API calls. Because neither component is self-evidently malicious in isolation, detectors that analyze each runtime independently fail to reconstruct the full attack semantics.

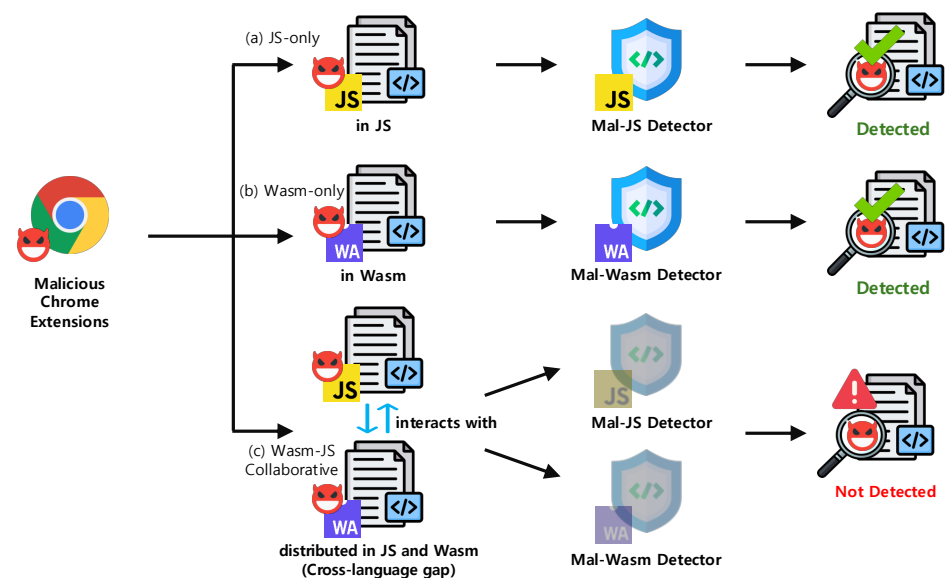


Figure 2. Malicious Extension Architectures and Their Detectability: (a) JS-only, (b) Wasm-only, and (c) Wasm–JS collaborative (cross-language gap).

The distinction is best illustrated by contrasting cryptomining implementations. In a Wasm-only miner, repetitive hash operations and abnormal memory access patterns are directly observable in the binary. In a collaborative miner, scheduling, server communication, and result exfiltration reside in JS, while Wasm exposes only arithmetic functions that are indistinguishable from a legitimate math library. The same principle applies to all attack types examined in this study.

Figure 3 concretizes this principle for the Forced Redirect scenario: the redirect target URL is computed and stored entirely within the Wasm binary, copied into JS via a linear memory read, and relayed to `chrome.tabs.update`—a three-stage chain in which the JS component is syntactically identical to the benign glue code shown in Figure 1, and therefore produces no signal for a JS-based analyzer.



Figure 3. Data flow across the Wasm–JS–Chrome API boundary in the Forced Redirect scenario. The redirect target URL is encoded in the Wasm binary and never appears in JS source, rendering it invisible to JS-based static analyzers.

3.3. Capability Model

To address RQ1, we enumerate the primitive capabilities available to a malicious extension author and organize them into three layers: JS & glue code, Wasm, and Chrome APIs. Table 1 provides the complete capability-to-attack mapping.

Table 1. Capabilities and Abuse Vectors of Wasm–JS Collaborative Malicious Extensions. Blue bold entries indicate Wasm-resident operations.

Capabilities	Sample API
JavaScript & Glue Codes	
Get Value	<code>document.getElementById('id').value</code>
Access Wasm Memory	<code>new Uint8Array(HEAPU8.buffer, ptr, size)</code>
Read File in Browser	<code>(MEMFS) FS.readFile(),</code> <code>(IDBFS) FS.mount(FS.filesystems.IDBFS, , '/idbfs') +</code> <code>FS.syncfs(true, callback) + FS.readFile()</code>
Event(e)	<code>addEventListener('click')</code>
Stop Event	<code>event.preventDefault()</code>
WebAssembly	
HTTP Request (URL)	<code>emscripten_fetch(URL)</code>
HTTP Response	<code>void onSuccess(emscripten_fetch_t *fetch) {...}</code>
Write & Save File in Browser	<code>(MEMFS) fopen('ShowDateTime.exe', 'wb')</code> <code>(IDBFS) fopen('/idbfs/ShowDateTime.exe', 'wb') +</code> <code>EM_ASM(FS.syncfs(false, callback))</code>
Encryption	<code>AES_CBC_encrypt_buffer(&ctx, buffer, data_len);</code>
Chrome API (JavaScript)	
Download	<code>chrome.downloads.download</code>
Message (key)	<code>chrome.runtime.sendMessage,</code> <code>chrome.runtime.onMessage.addListener</code>
Update Tab	<code>chrome.tabs.update(tabId, url: message.url)</code>
Read File in Extension	<code>chrome.runtime.getURL(key_logger.wasm)</code>
Proxy	<code>chrome.proxy.settings.set(), chrome.proxy.settings.get()</code>
Abuse Vector	
Action Models	
Adware	<code><iframe></code> → Read File in Extension (popup) → HTTP Request(URL) & HTTP Response → Access WebAssembly Memory → <code></code> <code><video></code>
Download Malicious File (Phishing File)	Write & Save File in Browser → Event('click') → Read File in Browser → Download
Download Malicious File (Hijacking Download URL)	<code><a></code> → Stop Event → HTTP Request(URL) & HTTP Response → Write & Save File in Browser → Read File in Browser → Download
Forced Redirect (General Redirect)	Proxy → Visit Any Site → Read File in Extension(.wasm) → HTTP Request(URL) & HTTP Response → Message(redirect) → Update Tab
Forced Redirect (Target-specific Phishing)	Proxy → Visit a Specific Site → Read File in Extension(.wasm) → HTTP Request(URL) & HTTP Response → Message(redirect) → Update Tab → Keylogger Credential Theft
Keylogger	Proxy → <code><iframe></code> → Read File in Extension(.wasm, glue code) → Event('input' 'keydown') → HTTP Request(ProxyServer)
Credential Theft	Event('submit') → Get Value → HTTP Request(Attacker's Server) & HTTP Response
Ransomware	<code><button></code> → User select File or Directory → Encryption

In the Action Models column, blue text refers to capability items listed in the upper part of the table, and red text refers to abuse-vector items listed in the lower part.

JS & glue code mediates between the browser environment and the Wasm module. Its responsibilities include retrieving DOM values (`document.getElementById.value`), accessing Wasm linear memory to extract module-computed data, managing Emscripten virtual filesystems (MEMFS for session-scoped storage; IDBFS for persistent IndexedDB-backed storage), and intercepting or suppressing browser events via `event.preventDefault()`.

WebAssembly encapsulates all logic that must remain opaque to static analyzers: target URLs for network requests and redirects, malicious file payloads encoded as binary arrays, cryptographic keys and encryption routines, and keylogging buffers. Outbound network communication from within Wasm is performed via `emscripten_fetch()`, which wraps the browser's XMLHttpRequest interface and allows HTTP requests to be issued without any URL appearing in JS source.

Chrome APIs provide privileged browser operations unavailable to plain JS: file downloads (`chrome.downloads`), tab manipulation (`chrome.tabs.update`), inter-component messaging (`chrome.runtime.sendMessage`), and system-wide proxy configuration (`chrome.proxy.settings`). Proxy access is critical for redirect-based attacks: by routing traffic through an attacker-controlled server configured with `Access-Control-Allow-Origin: *`, the extension bypasses the browser's CORS enforcement without triggering any manifest permission anomaly beyond the proxy entry.

These three layers form a unidirectional data pipeline in the collaborative architecture: sensitive values (URLs, keys, payloads) originate in the Wasm binary, cross the import/export boundary into JS glue code via linear memory reads, and are ultimately delivered to the browser through Chrome API calls. At no point in this pipeline does application-level semantics appear in the JS layer—only opaque byte buffers and pointer offsets—which is precisely why single-layer analyzers fail to reconstruct attack intent.

3.4. Attack Scenarios

Based on the capability model, we designed eight attack scenarios spanning six categories; malicious file delivery and forced redirection are each realized in two variants. Each follows the same execution pattern: (1) extension installation and activation, (2) a trigger from user interaction or a background timer, and (3) malicious behavior executed primarily within Wasm. For each scenario we describe the attacker's objective, the Wasm–JS division of responsibility, and the structural reason why the collaborative architecture resists detection.

Adware. The extension injects an `<iframe>` into every visited page and displays an attacker-specified advertisement. The target image URL is stored in Wasm and fetched via `emscripten_fetch()`; JS receives the image data through the Wasm memory interface and sets it as the `<img>` source in injected HTML. No hardcoded URL appears in JS, so the advertisement source is invisible to pattern-based static analysis.

Download Malicious File (Phishing File). A malicious file is encoded as a binary array inside the Wasm module and stored in an Emscripten MEMFS virtual filesystem. On trigger, JS reads the virtual file and initiates a download via `chrome.downloads`. The file payload is entirely absent from JS source.

Download Malicious File (Hijacking Download URL). The extension intercepts a legitimate download event via `event.preventDefault()`, retrieves a malicious substitution URL from Wasm through an HTTP request, and silently redirects the download. The user observes what appears to be a normal download interaction.

Forced Redirect (General). A proxy registered at extension startup routes all browser traffic through an attacker-controlled server. The redirect target URL is stored in Wasm and passed to `chrome.tabs.update` via messaging, causing every page visit to trigger a redirect.

Forced Redirect (Target-Specific Phishing). Functionally identical to the general redirect, but the trigger fires only when the user navigates to a predetermined domain (e.g., a banking site). Normal behavior on all other sites makes behavioral anomaly detection significantly harder. Because the extension behaves benignly on all domains except the target, dynamic analyzers that monitor short post-installation windows or sample a subset

of visited pages are unlikely to observe the redirect trigger, further reducing the probability of detection.

Keylogger. A keydown event listener captures each keystroke and passes it to Wasm, which buffers the input and transmits accumulated data to an attacker server URL encoded within the module. The proxy capability is reused to bypass CORS on the exfiltration channel. The exfiltration endpoint never appears in JS; a static analyzer inspecting the glue code would observe only a memory copy from a Wasm linear memory buffer to a network call, with no string literal identifying the destination.

Credential Theft. A `submit` event listener extracts form field values via `document.getElementById.value` and forwards them to an attacker URL stored in Wasm. This scenario requires no special manifest permissions, making it the most difficult attack to identify through permission-based static analysis. The absence of any required permission beyond the default makes this scenario indistinguishable from a legitimate form-analytics extension at the manifest level, and the destination URL hidden in Wasm prevents static detection at the code level as well.

Ransomware (User-Assisted). This scenario is a user-assisted, deceptive ransomware: the extension presents itself as a file-conversion utility to induce the user to grant file access. After the user selects target files via `window.showOpenFilePicker()`, Wasm encrypts them in-place using AES (via *tiny-AES-c* [32]) and overwrites the originals. Because Chrome's File System Access API requires explicit user consent, the extension cannot select targets autonomously—a meaningful constraint compared to traditional ransomware—and the attack instead relies on deceiving the user into selecting files through the picker; this does not, however, eliminate the threat against users who interact with a disguised utility. Browser-based ransomware leveraging the File System Access API and Wasm was demonstrated previously by RøB [33], a malicious *web application* that empirically showed conventional antivirus solutions struggle to detect such attacks. Our scenario differs in setting and framing: it operates within the Chrome extension ecosystem under MV3, and we treat the Wasm-JS split not merely as an implementation choice but as a systematic cross-language evasion vector against static extension detectors.

3.5. Implementation Notes

All PoC extensions were implemented using Emscripten 5.0.3 for Wasm compilation and tested on Chrome 144 running on Windows 11 Education. Each extension was loaded via Chrome's developer mode without submission to the Chrome Web Store. The Wasm modules were compiled with standard Emscripten optimization flags (`-O2`) and linked against the Emscripten Fetch API (`-sFETCH`) for network communication and the Emscripten filesystem (`-sFORCE_FILESYSTEM`) for virtual file I/O. The ransomware scenario additionally linked against the *tiny-AES-c* [32] library compiled as a Wasm static archive. Inter-runtime communication in all scenarios follows the same pattern: Wasm writes output to a designated region of linear memory, and JS glue code reads the result using a typed array view (e.g., `new Uint8Array(HEAPU8.buffer, ptr, len)`) before passing it to the appropriate Chrome API. All eight attack scenarios executed as intended across repeated trials, confirming that the Wasm-JS collaborative architecture is practically realizable within the constraints of the MV3 extension model.

Ethical considerations. All attack scenarios were demonstrated exclusively in a controlled local environment, and no PoC extension was submitted to the Chrome Web Store or run against real users or third-party services. Every component a scenario required—target pages, download sources, and network endpoints—was hosted locally, and each demonstration used only dummy data and harmless payloads. The keylogging and credential theft PoCs captured only synthetic keystrokes and dummy credentials entered on local

test pages; the adware, malicious-download, and forced-redirect PoCs injected placeholder advertisements, served harmless files, and redirected only to local destinations; and the ransomware PoC encrypted only designated test files within a local directory, using a retained key so that the operation was fully reversible and affected no real user data. No user data was collected, and no traffic left the local environment.

4. Evaluation

We evaluate the proposed attack model along two dimensions. First, we confirm that all eight PoC scenarios execute as intended (Section 4.2). Second, we assess whether four representative static malware detectors can identify the PoC extensions as malicious (Section 4.3), and analyze the structural reasons behind each detector’s failure or success (Section 4.4).

4.1. Experimental Setup

Detectors. We evaluate four tools that collectively represent the current state of extension malware detection:

- **Cujo** [13]: a static JS analyzer that extracts structural features from JS code. We retrained Cujo using 82,581 benign samples from the JS150K dataset [34] and 39,452 malicious samples from the Hynek Petrak dataset [35], yielding an approximately 2:1 benign-to-malicious ratio (as in Table 2). A strict 1:1 ratio would discard the majority of available benign data, while a full 3:1 ratio introduces class imbalance; the 2:1 ratio balances both concerns.
- **JStap** [14]: a JS analyzer supporting five code representations and two feature extraction methods (Ngrams and Value). Because Wasm cannot currently be represented as an AST, CFG, PDG, or PDG-DFG, we use the Token representation with both Ngrams and Value extractors. The same training dataset as Cujo was used.
- **WasmGuard** [20]: a machine learning model that directly analyzes Wasm binary files. We use the author-provided checkpoint that achieved 99% validation accuracy at epoch 15 [36].
- **VirusTotal** [21]: a multi-engine online scanner. For each of the eight PoC extensions we submitted the complete packaged extension as both .crx and .zip—not isolated .wasm or glue-code files—scanning each once, since VirusTotal deduplicates submissions by file hash. Its reports are public, whereas the samples, which contain only non-operational demonstration payloads, are downloadable only via VirusTotal Intelligence. Beyond 61–62 conventional signature- and ML-based antivirus engines, VirusTotal also exposes two experimental AI-based components (Code Insight and the crowdsourced Exodia Labs engine); the results in Section 4.3 concern the conventional engines, and we discuss the AI components in Section 5.3.

Table 2. Datasets for learning Cujo and JStap.

Class	Dataset	# Samples (Files)
Benign	JS150K	82,581
Malicious	Hynek Petrak	39,452

Retraining details. We partitioned this corpus into training, validation, and test sets in a 70/15/15 ratio (benign: 58,083/12,233/12,265; malicious: 27,616/5918/5918). For both detectors, we adapted the publicly available reference implementations to our corpus, retaining their default preprocessing pipelines and classifiers—a support-vector machine for Cujo and a random forest for JStap. Cujo’s feature space is a HashingVectorizer over code q-grams with its default dimensionality of $2^{20} = 1,048,576$; JStap uses the

Token representation, yielding 602 features (Token-Ngrams) and 24,912 (Token-Value). The nominal 15% test partition contains 12,265 benign and 5918 malicious files; after tool-specific parsing and feature-extraction losses, the effective evaluated test sets are 12,072/5088 (Cujo) and 12,259/5719 (JStap), the counts reported below.

Baseline detector performance. Before evaluating evasion, we confirmed that the re-trained JS analyzers perform competently on conventional JavaScript malware, so that any failure on our PoC reflects the cross-language blind spot rather than a degenerate model. On the held-out test sets, Cujo reached 96.8% accuracy (TP/FP/FN/TN = 5078/533/10/11,539) and JStap reached 99.8% under both extractors (e.g., 5692/0/27/12,259 for Ngrams); both recovered the malicious class at high recall (>99%). Notably, Cujo’s false-positive rate on ordinary benign JavaScript is modest ($\approx 4.4\%$), which makes the near-total false-positive rate it exhibits on toolchain-generated glue code (Section 4.4) a clear anomaly attributable to glue-code structure rather than a general tendency to over-flag. WasmGuard’s baseline competence is established by the author-provided checkpoint, which reached 99% validation accuracy [36].

4.2. Functional Verification

Prior to detection evaluation, we confirmed that all eight attack scenarios executed as intended across repeated trials in the local test environment. Each scenario successfully demonstrated its target behavior—forced advertisement injection, malicious file delivery, browser redirection, keystroke capture, credential exfiltration, and file encryption—within the permission constraints of the MV3 manifest model. Table 3 includes a functional confirmation column for each attack type.

Table 3. Malware Detection Results.

Attack Type	Detail	Cujo *	JStap		WasmGuard	VirusTotal
			Ngrams	Value		
Adware	N/A	○	×	×	×	×
Download Malicious File	Phishing File	○	×	×	×	×
	Hijacking Download URL	○	×	×	×	×
Forced Redirect	General Redirect	○	×	×	×	×
	Target-specific Phishing	○	×	×	×	×
Keylogger	N/A	○	×	×	×	×
Credential Theft	N/A	○	×	×	×	×
Ransomware	N/A	○	×	×	×	×

Legend: ○ = Detected; × = Not Detected; * = Requires verification;

4.3. Detection Evasion Results

Table 3 summarizes the detection outcome for each attack type across all four tools. The results reveal a consistent pattern: every PoC extension evaded all conventional detectors. JStap classified all JS glue code as benign with approximately 98% confidence under both the Token-Ngrams and Token-Value configurations. WasmGuard classified all Wasm modules as benign. Cujo flagged every PoC as malicious, but as we demonstrate in Section 4.4, this result reflects indiscriminate classification rather than genuine detection capability. VirusTotal returned no detections from its conventional engine set on any PoC sample: across all eight submissions, none of the 61–62 antivirus engines aggregated by the platform flagged the extension as malicious. VirusTotal additionally surfaces an experimental crowdsourced AI-based analysis whose behavior differs markedly from these conventional engines; because it bears on emerging rather than currently-deployed

detection, we examine it separately in Section 5.3. This shows that the cross-language attack architecture evades not only research-grade static analyzers but also the conventional signature- and ML-based engines that constitute the bulk of the deployed commercial ecosystem. These evasion outcomes are deterministic—each detector returns a fixed verdict for a given input—and are therefore stated as exact results rather than statistical estimates; sampling variation enters only in the real-world false-positive study below, for which we report confidence intervals.

4.4. Analysis of Detector Limitations

Cujo: false positives mask genuine detection failure. Although Cujo flagged all PoC extensions as malicious, this outcome cannot be interpreted as successful detection.

To investigate, we assembled a set of real-world Wasm-enabled extensions in three steps. First, from the Chrome Web Store extensions we had collected, we located those that use WebAssembly by searching their JavaScript for the `WebAssembly` keyword and verifying, within a 60-character window around each occurrence, that it corresponded to genuine module initialization; we used a keyword search rather than matching a specific loader because Wasm may be fetched externally or embedded as base64, which a loader-specific pattern would miss. Second, within these extensions we identified the JavaScript glue code by matching the unique identifiers that Wasm toolchains emit in their generated wrappers (e.g., Emscripten for C/C++, `wasm-bindgen` for Rust) together with the associated initialization code; the 71 extensions for which glue code could be located were retained. Third, we labelled each extension by its Chrome Web Store status as reported by `chrome-stats.com`: the 65 extensions still live were labelled benign, and the 6 that had been removed were labelled malicious (removal reasons: malware for three, bundling of unwanted software for one, a policy violation for one, and none stated for one). We then ran Cujo on the glue code of all 71 extensions.

We evaluate the outcome at two granularities (Table 4). At the extension level, treating an extension as malicious if any of its glue-code files is flagged, Cujo classified 49 of the 65 benign extensions (75%) as malicious, for a malicious-class precision of only 0.08 and an overall accuracy of 0.28. This over-flagging originates at the glue-code level: of the 123 benign glue-code files, Cujo flagged 93 (76%) as malicious, and of the 97 files it labelled malicious in total, only four were genuinely so (precision 0.04). To quantify the precision of these estimates, we report Wilson score 95% confidence intervals for the benign false-positive proportions. The extension-level rate of 49/65 (75.4%) has a 95% CI of [63.7%, 84.3%], and the glue-code-level rate of 93/123 (75.6%) has a 95% CI of [67.3%, 82.4%]; both lower bounds exceed 63%, so the conclusion that Cujo over-flags the large majority of benign Wasm-enabled extensions is robust to sampling variation. The malicious-class metrics rest on a much smaller sample (six extensions, seven glue-code files) and carry correspondingly wide intervals, so we base our conclusions on the benign false-positive rate. These results demonstrate that Cujo's detections are driven by the structural characteristics of toolchain-generated glue code—which it treats as a malicious pattern—rather than by any semantic understanding of intent, consistent with its indiscriminate flagging of every PoC in our suite. We note that these labels are proxies—a live listing does not guarantee benignity, and removal spans reasons beyond confirmed malware—but because the flagging is a structural response to toolchain-generated glue code (Section 4.1) rather than an assessment of intent, hidden maliciousness in a small fraction of the benign set cannot account for a flag rate of this magnitude. An operator relying on Cujo would therefore face an impractical false-positive rate on essentially any Wasm-enabled extension, rendering it unsuitable as a deployment-ready detector.

Table 4. Cujo classification of 71 real-world Wasm-enabled Chrome Web Store extensions at two granularities. At the extension level, an extension is flagged malicious if any of its glue-code files is flagged. Precision/recall are reported for the malicious class.

Unit (Benign/Mal.)	TP	FP	FN	TN	Acc.	Prec.	Rec.
Extension (65/6)	4	49	2	16	0.28	0.08	0.67
Glue code (123/7)	4	93	3	30	0.26	0.04	0.57

JStap: JS-only analysis cannot cross the language boundary. JStap’s failure is structural: its feature extraction pipeline operates exclusively on JS tokens and cannot parse or interpret Wasm bytecode. Because the PoC extensions expose only innocuous glue code in JS, JStap has no signal to act on. The approximately 98% benign confidence score suggests that Emscripten glue code closely resembles legitimate JS in the feature space on which JStap was trained, compounding the evasion.

WasmGuard: module-level analysis misses cross-language semantics. WasmGuard was designed to detect malicious Wasm modules by analyzing binary-level patterns. Its training set, moreover, was heavily augmented from a small base of collected malicious Wasm samples, which narrows the distribution of behaviors it has learned to recognize. In the Wasm–JS collaborative architecture, however, the Wasm module in isolation contains only computational primitives (HTTP fetch wrappers, encryption routines, data buffers) that are individually indistinguishable from legitimate library code. The malicious semantics emerge only from the interaction between the Wasm module and its JS context, which WasmGuard does not model. This confirms that even a Wasm-aware detector is insufficient when the threat model spans both runtimes.

5. Discussion

5.1. Feasibility of the Required Permission Set

A natural question arising from the attack model is whether the required permissions and manifest attributes are plausible in real-world extensions—that is, whether a malicious extension requesting these permissions would appear suspicious to a typical user or reviewer. To address this, we crawled the Chrome Web Store on 30 June 2025, collecting metadata for 165,314 active extensions, and measured, for each permission and manifest attribute required by our attack scenarios, how many extensions declare it and how many current users those extensions reach. This measurement is a lightweight, single-pass analysis of publicly available metadata and scales trivially to the full store; the computationally demanding problem is not measurement but defense—applying cross-language or dynamic analysis to every submission at store scale—which we discuss in Section 5.3.

The prevalence of the required permissions is high. The `downloads` permission, required for the Download Malicious File attack, is declared by 3996 extensions. The `tabs` permission, used in Forced Redirect attacks, appears in 35,533 extensions—roughly 21% of all extensions in the store. The `proxy` permission, required for Keylogger and Forced Redirect attacks, is declared by 826 extensions; notably, 316 of these also declare `tabs`, satisfying the full permission requirement for both attack types simultaneously.

The required manifest attributes are similarly common. The `content_scripts` (CS) and `web_accessible_resources` (WAR) combination required for Adware is present in 36,013 extensions; the `host_permissions` (HP) attribute required for Download Malicious File is declared by 23,765; the most demanding configuration—CS, background scripts (BG), WAR, and HP combined, as required for Forced Redirect and Keylogger—is satisfied by 9545 extensions; and `web_accessible_resources` alone, required for Ransomware, is used by 43,045.

Tables 5 and 6 additionally report the distribution of current active users across the extensions in each group. Two points bear emphasis. First, these are current active-user counts rather than cumulative install counts, which the Chrome Web Store metadata does not expose; the distributions are computed over the extensions in each group that report such a count (column *N*). Second, the distributions are strongly right-skewed—for example, extensions declaring tabs have a median of only 103 current users but a maximum of 337 million—so we report quartiles rather than aggregate totals, which a handful of very large extensions would otherwise dominate. Regardless of scale, the central point stands: every permission and manifest attribute required by our attack model is already in widespread legitimate use, so a malicious extension requesting any of these configurations would not stand out from the existing extension population on the basis of its permission profile alone. The proposed attacks are therefore not only technically feasible but also practically concealable within the current extension ecosystem.

Table 5. Distribution of current active users across extensions declaring each required permission. *N* is the number of such extensions for which a current-user count is available on the Chrome Web Store; values are current active users, not cumulative install counts.

Permission	<i>N</i>	Q1	Median	Q3	Max
downloads	3178	44	347	3000	337,000,000
tabs	26,436	16	103	1000	337,000,000
proxy	648	131	1000	10,000	19,000,000
tabs, proxy	240	116	1000	20,000	19,000,000

Table 6. Distribution of current active users across extensions declaring each required manifest attribute. *N* and the reported values follow the same convention as Table 5.

Attribute	<i>N</i>	Q1	Median	Q3	Max
BG	73,293	18	117	1000	337,000,000
CS	60,454	15	83	717	337,000,000
HP	17,922	24	185	2000	337,000,000
WAR	32,018	23	158	1000	337,000,000
CS, WAR	26,749	23	164	1000	337,000,000
CS, BG, WAR, HP	7091	44	431	5000	337,000,000

5.2. Implications for Threat Scale

The prevalence data above carries a direct implication for potential attack scale, though one that must be read against the skewed user distribution. Consider the 316 extensions that already hold both tabs and proxy, the configuration required for both the Keylogger and Forced Redirect attacks. Among the 240 of these with an available user count, the distribution is heavily right-skewed: the median reaches roughly 1000 current users and the third quartile roughly 20,000, while the most popular reaches nearly 19 million (Table 5). A malicious extension in this category is therefore unlikely to reach hundreds of thousands of users at median adoption; the heavy right tail, however, shows that such reach is attainable—an extension at the upper quartile already exposes tens of thousands of users, and the tail extends into the millions. If distributed under the guise of a legitimate utility—a common social-engineering vector—even a mid-tail malicious extension would expose a substantial victim pool. Given that none of the evaluated static detectors genuinely detects these attacks, that exposure could persist over a prolonged window.

5.3. Directions for Defense

The evasion demonstrated across all tested detectors reflects a single structural deficiency established in Section 4.4: each detector reasons about one execution layer in isolation and cannot trace semantics across the Wasm–JS boundary into Chrome API calls. This is a property of the single-layer paradigm rather than of individual tools, and it holds whether the analysis is static or dynamic. Effective detection therefore requires reasoning over the Wasm bytecode in conjunction with its JS context—the direction we develop below.

Browser-vendor countermeasures. The principal vendor response to extension abuse is Manifest V3, which tightens permission scoping and prohibits remotely hosted code. MV3 raises the bar for several attack classes but permits both Wasm modules and their JS glue code; moreover, although its default content-security policy requires an extension to declare `wasm-unsafe-eval` to execute WebAssembly, this declaration is common among legitimate Wasm-using extensions and does not distinguish malicious use. Chrome Web Store review additionally incorporates behavioral signals, yet malicious extensions have been shown to pass it even under MV3 [6,30]. To our knowledge, no deployed vendor mechanism reconstructs the Wasm–JS cross-language data flow that our attacks exploit, so the gap persists despite these measures.

Dynamic and behavioral analysis, which we do not evaluate experimentally in this work, can in principle observe the runtime effects that ultimately surface as Chrome API calls; accordingly, we reason about its limitations analytically rather than empirically, and do not claim that placing logic in Wasm conceals these effects at runtime. Its limitations are of a different kind. Attacks gated behind specific victim conditions—such as the target-specific phishing scenario (Section 3.4)—do not fire during the bounded observation windows or page samples that practical monitors employ. Applying comprehensive dynamic analysis to every submission and update at Chrome Web Store scale would impose a cost that static pre-filtering is specifically designed to avoid. And even when a malicious effect is observed, a single-layer monitor cannot reconstruct the cross-language data flow that produced it, leaving the analyst without the provenance needed to separate a Wasm-driven attack from a benign extension exhibiting the same surface behavior. These constraints motivate detection that reasons across the Wasm–JS boundary rather than within either runtime alone.

Our results point to several directions for improving extension security against Wasm-assisted attacks.

Cross-language analysis. The fundamental limitation shared by all tested detectors is their inability to track data and control flow across the Wasm–JS boundary. A defense-oriented tool would need to model the full execution context: Wasm imports and exports, glue-code data marshaling, and Chrome API call sequences. Two implementation strategies are plausible. A static approach could lift the Wasm binary to a readable form (e.g., the WebAssembly text format) and construct a cross-language control- and data-flow graph spanning the lifted Wasm and its JS glue code, remaining within the low-cost pre-filtering regime that platform-scale screening requires. Alternatively, dynamic taint tracking instrumented at the Wasm–JS interface offers higher precision at the cost of non-trivial runtime overhead. JWBinder [31] demonstrates the value of the cross-language approach for general web malware, but does not model the Chrome extension APIs central to our scenarios.

Permission-aware behavioral monitoring. Since the required permissions are indistinguishable from those of legitimate extensions, static permission analysis alone is insufficient. Runtime monitoring that correlates permission usage with observed behavior—flagging, for example, an extension that holds proxy and tabs permissions and initiates Wasm-to-

proxy communication shortly after installation—could provide a practical detection signal without requiring full cross-language analysis.

AI-assisted analysis. A further, exploratory direction is suggested by the behavior of emerging AI-based detection. We report the following as a non-systematic observation—based on a single experimental engine and our eight PoC samples rather than a controlled evaluation—intended to motivate future work rather than to characterize AI-based detection in general. Beyond its conventional engines, VirusTotal exposes an experimental crowdsourced AI analysis (Exodia Labs), which flagged seven of our eight PoC samples as malicious and, through its natural-language summaries, accurately described behaviors such as Wasm module instantiation, XMLHttpRequest-based communication, and iframe-based ad injection. This stands in sharp contrast to the conventional engines and indicates that AI-assisted reasoning can partially close the static blind spot. Its single failure, however, delineates the current boundary: the ransomware PoC was dismissed on the basis of its encryption primitive rather than recognized as an extension that irreversibly encrypts and overwrites user files. The analysis reasoned about an isolated component without reconstructing the behavioral intent; tellingly, the ransomware is also the only scenario producing no outbound communication, leaving no network signal to observe. Current AI-based detection thus raises the bar through observable behavioral and network cues but does not yet model the Wasm–JS semantics that define the threat. Whether such analysis can be developed into a reliable cross-language detector, and how robust it would be to adversarial evasion, remains an open question.

5.4. Limitations

This study has several limitations. First, the PoC extensions were evaluated in a local environment and were not subjected to real-world distribution or Chrome Web Store review; our claims concern the detection capability of the evaluated tools, not the outcome of the platform’s full vetting pipeline. Second, our evaluation covers four representative static detectors, and other approaches may behave differently. Third, the detector training corpora (JS150K and Hynek Petrak) are general JavaScript datasets rather than extension-specific ones; because the code our attack exposes is ordinary Emscripten-generated glue, a JS-trained detector remains an appropriate instrument for testing whether the JavaScript surface reveals the attack, but detectors and datasets tailored to the extension setting are left to future work. Fourth, our observations on AI-assisted analysis rest on a single experimental crowdsourced engine and one missed sample; we therefore do not claim that AI-based detection is generally evadable, only that its current behavior reveals a semantic gap worth further study. Finally, the ransomware PoC used a statically embedded AES key for demonstration purposes, whereas a realistic variant would derive or retrieve keys dynamically; this simplification does not affect the cross-language evasion property, which concerns the concealment of logic rather than the strength of the cryptography.

6. Related Work

This paper investigates a previously unexplored threat surface at the intersection of WebAssembly and Chrome extensions. We situate our work within three lines of related research: Wasm-based evasion techniques, malicious Wasm detection, and browser extension security.

6.1. Wasm-Based Evasion of Malware Detectors

Wobfuscator [19] demonstrated that converting the malicious logic of JS malware into Wasm dramatically reduces detection rates by JS-based analyzers such as Cujo [13], JaSt [37], and JStap [14], while having limited effect on AST-based detectors such as Zozzle [38]

that retain syntactic signals from surrounding JS code. Cabrera-Arteaga et al. [39] further showed that Wasm binaries can be diversified through semantic-preserving transformations to evade cryptomining detectors, demonstrating that binary-level evasion is an active and evolving threat. More recently, WASMixer [40] introduced the first general-purpose Wasm binary obfuscator, applying data- and code-level obfuscation that hinders state-of-the-art static analyzers, further confirming that binary-level evasion of Wasm tooling is an active research front.

6.2. Malicious WebAssembly Detection

Research on malicious Wasm has focused primarily on browser-based cryptojacking [41,42]. Dedicated detectors including CoinSpy [43], MineThrottle [44], and MINOS [45] identify mining workloads by their binary-level patterns, such as high arithmetic density and repetitive hash operations. WasmGuard [20] generalizes beyond these mining-specific detectors with a machine-learning classifier that operates directly on raw Wasm bytecode. These detectors share a common assumption: malicious behavior is self-contained within the Wasm module and detectable from its binary structure alone. JWBinder [31] reconstructs JS–Wasm data flow for malware detection on general web pages, reporting substantially improved detection over monolingual scanners.

6.3. Browser Extension Security

The security of browser extensions has been studied from multiple angles. Eriksson et al. [46] found that approximately 15,000 of 130,000 Chrome extensions are malicious or pose security risks, underscoring the scale of the problem. Hsu et al. [1] provide a holistic view of the Chrome Web Store, showing that security-noteworthy extensions affect nearly 350 million users and persist in the store for years. Aggarwal et al. [4] characterize spying extensions that capture users' online activity and exfiltrate it to third-party domains, demonstrating detection via RNN-based API-call-sequence analysis. Jagpal et al. [17] proposed Hulk, which detects malicious extensions through dynamic behavioral analysis of DOM manipulation and network activity. Fass et al. [15] introduced DoubleX, a static analyzer that constructs an enriched control flow graph over extension JS to identify suspicious data flows between privileged APIs and external endpoints. Singh et al. [30] demonstrated that malicious extensions implementing behaviors such as keylogging and cookie theft can pass Chrome Web Store review even under MV3, and Rosenzweig et al. [16] showed that supervised machine learning, even when trained on thousands of malicious samples, faces substantial practical difficulty in reliably detecting malicious extensions at Chrome Web Store scale.

These studies share a JS-centric analytical model: they reason about extension behavior exclusively through JS artifacts. DoubleX and Hulk would analyze only the glue code exposed by our PoC extensions, which contains no suspicious data flows or behavioral anomalies. Our work demonstrates that this analytical blind spot is practically exploitable.

Arcanum [18] is the most closely related work in terms of methodology, applying dynamic taint tracking to detect privacy-violating data flows in extension JS. Its focus on JS-only taint propagation, however, means that data originating from or destined for Wasm modules would not be tracked across the language boundary—precisely the gap our attack model exploits.

6.4. Novelty and Positioning

Detection of malicious extensions has historically relied on static analysis of JavaScript, which constitutes the bulk of extension code. Wobfuscator [19] undermined this assumption by translating malicious JavaScript logic into WebAssembly, showing that a JS-only analyzer can be evaded; the malicious logic still originates in JS and is merely transformed,

whereas in our threat model it is authored natively in Wasm and the surrounding JavaScript carries no malicious signal at all. The closest conceptual framing is JWBinder [31], which defines JavaScript–WebAssembly multilingual malware (JWMM) and reconstructs the cross-language data flow so that monolingual JS detectors can be applied with improved detection; this validates cross-language reasoning as the appropriate defensive direction, but it is evaluated on general web pages and does not model the Chrome extension environment or the privileged APIs on which our scenarios depend. WasmGuard [20] represents the state of the art in directly detecting malicious Wasm through machine learning over raw bytecode, yet it classifies a module in isolation under the assumption that maliciousness is self-contained in the binary—an assumption our attacks invalidate, since each Wasm module exposes only benign-looking computational primitives and the malicious semantics reside in the Wasm–JS interaction. Prior browser-extension security studies [15,17,18,46] likewise reason about extension behavior exclusively through JavaScript artifacts and therefore inherit the same blind spot. The novelty of this work is the synthesis of these gaps: to our knowledge, we are the first to characterize the Wasm–JS collaborative architecture as an attack-construction paradigm within the Chrome extension ecosystem, to realize it as a suite of MV3 proof-of-concept extensions that simultaneously evade JS-based, Wasm-aware, and deployed commercial detectors, and to quantify—through an analysis of 165,314 live extensions—that the capabilities these attacks require are already in ubiquitous legitimate use.

7. Conclusions

This paper investigated the feasibility and detectability of malicious Chrome extensions that exploit the Wasm–JS collaborative architecture. By embedding malicious logic within WebAssembly and confining JavaScript to minimal glue code, we demonstrated that the core of each attack can be rendered invisible to the static analyzers we evaluate.

We designed eight attack scenarios spanning six categories—adware, malicious file delivery, forced redirection, keylogging, credential theft, and ransomware—and implemented each as a functioning PoC under the MV3 manifest model. Evaluation against four representative static detectors showed that none achieved genuine detection: three classified the glue code as benign or raised no alerts, while the fourth flagged every sample only through systematic false positives on toolchain-generated glue code, confirmed by a controlled experiment on real-world CWS extensions. Analysis of 165,314 live CWS extensions further confirmed that every permission required by our attack model is already in widespread legitimate use, meaning a Wasm-assisted malicious extension would not be flagged by permission-based screening alone.

Closing this gap will require detectors that reason across the Wasm–JS boundary—through cross-language analysis, whether static or dynamic, or permission-aware behavioral monitoring—rather than within either runtime in isolation.

Author Contributions: Conceptualization, J.P.; Methodology, Y.M. and J.P.; Software, Y.M. and M.H.; Validation, Y.M.; Investigation, Y.M.; Data curation, Y.M. and M.H.; Writing—original draft, Y.M.; Writing—review & editing, J.P.; Visualization, Y.M.; Supervision, J.P.; Project administration, J.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported in part by a grant from Kyung Hee University in 2023 (KHU-20233246), and by the MSIT (Ministry of Science and ICT), Korea, under the Convergence Security Core Talent Training Business Support Program (RS-2023-00266615) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

Data Availability Statement: The proof-of-concept extensions were implemented with non-operational demonstration payloads and executed only in a local research environment; samples

submitted to VirusTotal for scanning contain only non-operational payloads and are retrievable only via VirusTotal Intelligence. To prevent misuse, we do not publicly release the extensions or their source code; these are available to verified researchers upon reasonable request to the corresponding author. The detector training data are publicly available (the JS150K [34] and Hynek Petrak [35] datasets), and the extension permission statistics were derived from publicly accessible Chrome Web Store metadata.

Acknowledgments: The authors used AI-based tools (e.g., large language model assistants) solely for language editing—specifically translation and grammar checking—during manuscript preparation. AI tools were not used in the design of the study, analysis, or the generation of any scientific results or conclusions reported in this work. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Hsu, S.; Tran, M.; Fass, A. What is in the Chrome Web Store? In *Proceedings of the 19th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Singapore, July 2024; ACM: New York City, NY, USA, 2024.
2. Kim, Y.M.; Lee, B. Extending a Hand to Attackers: Browser Privilege Escalation Attacks via Extensions. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, August 2023; USENIX Association: Berkeley, CA, USA, 2023.
3. 100+ Fake Chrome Extensions Found Hijacking Sessions, Stealing Credentials, Injecting Ads. Available online: <https://thehackernews.com/2025/05/100-fake-chrome-extensions-found.html> (accessed on 25 March 2026).
4. Aggarwal, A.; Viswanath, B.; Zhang, L.; Kumar, S.; Shah, A.; Kumaraguru, P. I Spy with My Little Eye: Analysis and Detection of Spying Browser Extensions. In *Proceedings of the European Symposium on Security and Privacy (EuroS&P)*, London, United Kingdom, April 2018; IEEE: Washington, DC, USA, 2018.
5. Saini, A.; Gaur, M.S.; Laxmi, V.; Conti, M. Colluding browser extension attack on user privacy and its implication for web browsers. *Comput. Secur.* **2016**, *63*, 14–28.
6. Nayak, A.; Khandelwal, R.; Fernandes, E.; Fawaz, K. Experimental Security Analysis of Sensitive Data Access by Browser Extensions. In *Proceedings of the 33rd International World Wide Web Conference (WWW)*, Singapore, May 2024; ACM: New York City, NY, USA, 2024.
7. Popular Chrome Extensions Leak API Keys, User Data via HTTP and Hard-Coded Credentials. Available online: <https://thehackernews.com/2025/06/popular-chrome-extensions-leak-api-keys.html> (accessed on 25 March 2026).
8. Researchers Reveal 18 Malicious Chrome and Edge Extensions Disguised as Everyday Tools. Available online: <https://www.infosecurity-magazine.com/news/18-malicious-chrome-edge-extensions> (accessed on 25 March 2026).
9. Chrome Extension Active User Count. Available online: <https://chrome-stats.com/chrome/stats> (accessed on 25 March 2026).
10. Dozens of Chrome Browser Extensions Hijacked by Data Thieves. Available online: <https://www.infosecurity-magazine.com/news/chrome-browser-extensions-hijacked> (accessed on 25 March 2026).
11. DeKoven, L.F.; Savage, S.; Voelker, G.M.; Leontiadis, N. Malicious Browser Extensions at Scale: Bridging the Observability Gap between Web Site and Browser. In *Proceedings of the USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, Vancouver, Canada, August 2017; USENIX Association: Berkeley, CA, USA, 2017.
12. Security Experts Flag Chrome Extension Using AI Engine to Act Without User Input. Available online: <https://www.infosecurity-magazine.com/news/chrome-extension-ai-engine-act-mcp> (accessed on 25 March 2026).
13. Rieck, K.; Krueger, T.; Dewald, A. Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, Austin, TX, December 2009; ACM: New York City, NY, USA, 2009.
14. Fass, A.; Backes, M.; Stockand, B. JStap: A Static Pre-Filter for Malicious JavaScript Detection. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, San Juan, Puerto Rico, December 2019; ACM: New York City, NY, USA, 2019.
15. Fass, A.; Somé, D.F.; Backes, M.; Stock, B. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Seoul, South Korea, November 2021; ACM: New York City, NY, USA, 2021.
16. Rosenzweig, B.; Dalla Valle, V.; Apruzzese, G.; Fass, A. It's Not Easy: Applying Supervised Machine Learning to Detect Malicious Extensions in the Chrome Web Store. *ACM Trans. Web (TWEB)* **2025**, *20*, 1–37. <https://doi.org/10.1145/3770852>.
17. Jagpal, N.; Dingle, E.; Gravel, J.P.; Mavrommatis, P.; Provos, N.; Rajab, M.A.; Thomas, K. Trends and Lessons from Three Years Fighting Malicious Extensions. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, August 2015; USENIX Association: Berkeley, CA, USA, 2015.

18. Xie, Q.; Murali, M.V.K.; Pearce, P.; Li, F. Arcanum: Detecting and Evaluating the Privacy Risks of Browser Extensions on Web Pages and Web Content. In *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, August 2024; USENIX Association: Berkeley, CA, USA, 2024.
19. Romano, A.; Lehmann, D.; Pradel, M.; Wang, W. Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2022; IEEE: Washington, DC, USA, 2022.
20. Sun, Y.; Chen, H.; Fu, Z.; Lv, W.; Liu, Z.; Liu, H. WasmGuard: Enhancing Web Security through Robust Raw-Binary Detection of WebAssembly Malware. In *Proceedings of the 31st International World Wide Web Conference (WWW)*, Virtual Conference, April 2022; ACM: New York City, NY, USA, 2022.
21. VirusTotal. Available online: <https://virustotal.com> (accessed on 25 March 2026).
22. Chrome Extension with 100,000 Users Caught Pushing Cryptocurrency Miner. Available online: <https://www.bleepingcomputer.com/news/security/chrome-extension-with-100-000-users-caught-pushing-cryptocurrency-miner/> (accessed on 25 March 2026).
23. Musch, M.; Wressnegger, C.; Johns, M.; Rieck, K. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *Proceedings of the 16th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Gothenburg, Sweden, June 2019; Springer: Cham, Switzerland, 2019.
24. Manifest V2 Support Timeline. Available online: <https://developer.chrome.com/docs/extensions/develop/migrate/mv2-deprecation-timeline/> (accessed on 25 March 2026).
25. WebAssembly Main Site. Available online: <https://webassembly.org/> (accessed on 25 March 2026).
26. WebAssembly Feature Status in Modern Web Browser. Available online: <https://webassembly.org/features/> (accessed on 25 March 2026).
27. emscripten | Emscripten is a Complete Compiler Toolchain to WebAssembly, Using LLVM, with a Special Focus on Speed, Size, and the Web Platform. Available online: <https://emscripten.org/> (accessed on 25 March 2026).
28. Lehmann, D.; Kinder, J.; Pradel, M. Everything Old is New Again: Binary Security of WebAssembly. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual Conference, August 2020; USENIX Association: Berkeley, CA, USA, 2020.
29. Hilbig, A.; Lehmann, D.; Pradel, M. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the 30th International World Wide Web Conference (WWW)*, Virtual Conference, April 2021; ACM: New York City, NY, USA, 2021.
30. Singh, S.; Varshney, G.; Singh, T.K.; Mishra, V. A Study on Malicious Browser Extensions in 2025. *arXiv* **2025**, arXiv:2503.04292.
31. Xia, Y.; He, P.; Zhang, X.; Liu, P.; Ji, S.; Wang, W. Static Semantics Reconstruction for Enhancing JavaScript-WebAssembly Multilingual Malware Detection. In *Proceedings of the 28th European Symposium on Research in Computer Security (ESORICS)*, Hague, The Netherlands, September 2023; Springer: Cham, Switzerland, 2023.
32. Tiny AES in C. Available online: <https://github.com/kokke/tiny-AES-c/> (accessed on 25 March 2026).
33. Oz, H.; Aris, A.; Acar, A.; Tuncay, G.S.; Babun, L.; Uluagac, S. RøB: Ransomware over Modern Web Browsers. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, August 2023; USENIX Association: Berkeley, CA, USA, 2023.
34. JS150K—Benign JavaScript Datasets. Available online: <https://www.sri.inf.ethz.ch/js150> (accessed on 25 March 2026).
35. Hynek Petrak Javascript Malware Collection—Malware JavaScript Datasets. Available online: <https://github.com/HynekPetrak/javascript-malware-collection> (accessed on 25 March 2026).
36. WasmGuard Validation Accuracy 0.99 ckpt. Available online: <https://github.com/Yuxia-Sun/WasmGuard/tree/main/model/ckpt/classifier> (accessed on 25 March 2026).
37. Fass, A.; Krawczyk, R.P.; Backes, M.; Stockand, B. JaSt: Fully Syntactic Detection of Malicious(Obfuscated) JavaScript. In *Proceedings of the 15th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Saclay, France, June 2017; Springer: Cham, Switzerland, 2017.
38. Curtsinger, C.; Livshits, B.; Seifert, B.Z.C. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, August 2011; USENIX Association: Berkeley, CA, USA, 2011.
39. Cabrera-Arteaga, J.; Monperrus, M.; Toady, T.; Baudry, B. WebAssembly diversification for malware evasion. *Elsevier Comput. Secur.* **2023**, *131*, 103296.
40. Cao, S.; He, N.; Guo, Y.; Wang, H. WASMixer: Binary Obfuscation for WebAssembly. In *Proceedings of the 29th European Symposium on Research in Computer Security (ESORICS)*, Bydgoszcz, Poland, September 2024; Springer: Cham, Switzerland, 2024.
41. Musch, M.; Wressnegger, C.; Johns, M.; Rieck, K. Thieves in the Browser: Web-based Cryptojacking in the Wild. In *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES)*, Canterbury CA, United Kingdom, August 2019; ACM: New York City, NY, USA, 2019.
42. Musch, M.; Wressnegger, C.; Johns, M.; Rieck, K. Web-based Cryptojacking in the Wild. *arXiv* **2018**, arXiv:1808.09474.
43. Kelton, C.; Balasubramanian, A.; Raghavendra, R.; Srivatsa, M. Browser-Based Deep Behavioral Detection of Web Cryptomining with CoinSpy. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2020; The Internet Society: Reston, VA, USA, 2020.

44. Bian, W.; Meng, W.; Zhang, M. MineThrottle: Defending against Wasm In-Browser Cryptojacking. In *Proceedings of the 29th International World Wide Web Conference (WWW), Taipei, Taiwan, April 2020*; ACM: New York City, NY, USA, 2020.
45. Naseem, F.; Aris, A.; Babun, L.; Tekiner, E.; Uluagac, A.S. MINOS: A Lightweight Real-Time Cryptojacking Detection System. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS), Virtual Conference, February 2021*; The Internet Society: Reston, VA, USA, 2021.
46. Eriksson, B.; Picazo-Sanchez, P.; Sabelfeld, A. Hardening the security analysis of browser extensions. In *Proceedings of the 2022 ACM Symposium on Applied Computing (SAC), Virtual Conference, April 2022*; ACM: New York City, NY, USA, 2022.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.