

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

CHAIR OF PROGRAMMING LANGUAGES AND ARTIFICIAL INTELLIGENCE



Reevaluating Message-Passing Vulnerabilities in Chrome Extensions: A Replication Study of CoCo

Tudor Teofănescu

Master's Thesis Computer Science

Supervisor: Prof. Dr. Johannes Kinder

Advisor: Matias Gobbi

Submission Date: November 10, 2025

Declaration

I hereby confirm that this Master's Thesis is my own work, and that I have used no sources or aids other than those indicated. Passages that are taken either verbatim or in essence from other works are clearly marked as such and referenced accordingly.

Munich, November 10, 2025
Tudor Teofănescu

Acknowledgments

I would like to express my heartfelt gratitude to everyone who has supported me throughout the journey of this thesis. Special thanks to my advisor, colleagues, and friends for their invaluable guidance and encouragement.

Finally, I wish to acknowledge the tools and prior research that made this work possible. I thank Anthropic for developing Claude Code, which greatly facilitated the analysis of large-scale extension datasets and the development of research infrastructure. I also extend my appreciation to the authors of CoCo for their foundational work in Chrome extension security analysis, upon which this research builds.

Abstract

Browser extensions enhance web functionality for millions of users. However, their privileged access to sensitive browser capabilities makes them attractive attack targets. Extensions use content scripts to interact with untrusted web pages while background scripts execute powerful operations. This architectural split creates privilege escalation risks: malicious web content can exploit message-passing interfaces between these components to trigger sensitive operations. Static analysis tools detect such vulnerabilities by tracking information flows from attacker-controlled sources to privileged sinks, yet their precision on real-world extensions and the feasibility of automated validation remain open questions.

We applied CoCo static analysis to 197,088 Chrome Web Store extensions, flagging 1,887 (0.96%) as potentially vulnerable. Manual analysis of 239 sampled extensions established 36.4% precision (90% CI: 32–42%), with three dominant false positive patterns accounting for 74.3% of misclassifications: trusted developer backend communications, incomplete storage exploitation chains, and attacker parameters to benign operations.

We implemented LLM-based validation using Claude Sonnet 4.5 agents across all 1,887 flagged extensions. Comparison with manual ground truth revealed 81.6% agreement but 43.7% false negative rate. LLMs effectively filtered false positives (96.1% specificity) but missed many genuine vulnerabilities (55.2% sensitivity), with systematic failures in cross-file dependency tracking (31.6%) and threat model reasoning (26.3%).

Multi-tool agreement between CoCo and DoubleX yielded 89.2% precision on 74 jointly-flagged extensions—a 2.5× improvement over CoCo alone. These findings demonstrate that multi-tool agreement filters tool-specific false positives, while LLM validation serves as effective first-pass triage rather than standalone verification.

Contents

1	Introduction	1
2	Background	4
2.1	Chrome Browser Extensions	4
2.1.1	Manifest and Permission Model	5
2.1.2	Isolation Architecture and Communication Patterns	5
2.2	CoCo Vulnerability Detection	6
2.2.1	Analysis Pipeline	7
2.2.2	Threat Model and Vulnerability Types	8
2.2.3	Analysis Output	10
2.3	Related Static Analysis Tools	11
2.3.1	DoubleX: Hybrid Taint Tracking	12
2.3.2	CodeX: Contextual Flow Tracking	12
3	Methodology	13
3.1	Threat Model	13
3.1.1	Adversary Capabilities	13
3.1.2	Privileged Sinks	14
3.1.3	Exploitable Impact Criteria	14
3.1.4	Domain Restrictions Are Not Security Boundaries	15
3.2	Vulnerability Validation Methodology	15
3.2.1	Four-Step Validation Approach	16
3.2.2	Classification Criteria	16
3.2.3	Manual Validation Approach	17
3.2.4	LLM-Based Validation Approach	17
4	Implementation	19
4.1	CoCo Execution at Scale	19
4.1.1	CoCo Invocation Command	19
4.2	CoCo Code Problems and Custom Orchestration	20
4.2.1	CoCo Timeout Mechanism Failure	20
4.2.2	CoCo Parallel Mode Lacks Progress Persistence	20
4.3	Mass Processor Implementation	21
4.4	Categorization Script for CoCo Report	21
4.5	LLM-Based Analysis Workflow	23
4.5.1	Methodology Encoding for LLM Analysis	23
4.5.2	Batch Execution Workflow	23

4.5.3	Multi-Batch Progress Tracking	24
4.5.4	Agent Runtime Metrics	24
5	Results	26
5.1	Static Analysis Detection Results	26
5.1.1	Dataset Categorization	26
5.1.2	Vulnerability Detection Statistics	27
5.2	CoCo 2021 Baseline Validation	28
5.2.1	Manual Analysis Results	28
5.2.2	LLM Validation on 2021 Dataset	28
5.3	CoCo Precision (RQ1)	29
5.3.1	Manual Validation Results	29
5.3.2	Common True Positive Patterns	29
5.3.3	Common False Positive Patterns	30
5.4	LLM-Based Validation (RQ2)	31
5.4.1	Classification Results	32
5.4.2	Agreement with Manual Ground Truth	32
5.4.3	LLM Classification Errors	32
5.5	Multi-Tool Agreement (RQ3)	33
5.5.1	Overlap Analysis	33
5.5.2	Precision Improvement	34
6	Conclusion	35

Chapter 1

Introduction

Browser extensions enhance web browser functionality, but at the same time their privileged access to sensitive browser functionalities create significant security risks. Extensions can access cookies, browsing history, and page content across websites, while message-passing interfaces expose attack surfaces that malicious webpages or other extensions can exploit. Static analysis tools like CoCo [16] detect these vulnerabilities at scale, but their precision remains unvalidated on a large scale. In the following we will explore how many detections of such vulnerabilities are found by CoCo in the 2025 dataset and explore the nature of their exploitability from an untrusted source.

Recent studies reveal concerning trends in software security: 76% of developers are using or planning to use Large Language Model (LLM)-based coding tools [14], with 24.2% of LLM-generated JavaScript containing security weaknesses [5]. These findings underscore the growing urgency of thorough security validation for browser extensions, as the proliferation of LLM-assisted development may introduce further vulnerabilities at scale. Establishing precision under refined threat models and developing scalable validation approaches is therefore more critical than ever to protect users and improve the overall security posture of extension ecosystems. LLM-based validation represents a promising approach for automated security assessment, as it can already understand code and the task remains to configure the LLM or prompt to better find these exploits. The current changes for developers as well as an expanding ecosystem make such systematic validation capabilities particularly timely and necessary.

This thesis addresses this challenge through two complementary approaches. First, we measure detection precision under a refined threat model through systematic manual analysis of a statistically representative sample, developing explicit exploitability criteria that refine threat model boundaries for hardcoded backends and storage-based flows. Second, we investigate whether Large Language Model (LLM)-based agents can automate security analysis at scale, measuring their agreement with manual ground truth and identifying systematic limitations in automated vulnerability validation.

Research Questions RQ1: What is the precision of CoCo detections under a refined threat model?

We establish ground truth through manual security analysis of 239 systematically sampled extensions from the 1,887 extensions flagged by CoCo across the complete Chrome Web Store dataset crawled on 06.02.2025. Using a four-step validation methodology (Section 3.2) with explicit exploitability criteria, we classify each detection as exploitable (true positive) or benign (false positive). Our analysis reveals CoCo achieves 36.4% precision, with 90% confidence, thus

the true precision falls between 32% and 42%. Three dominant false positive patterns: trusted developer backend communications, storage operations lacking complete exploitation chains, and attacker-controlled parameters to benign APIs. These findings establish the first large-scale precision measurement for CoCo and identify specific threat model boundaries that account for the majority of false positives.

RQ2: How effectively can LLM-based agents validate static analysis vulnerability reports for Chrome extensions?

We develop an LLM-based validation system using Claude Sonnet 4.5 agents that applies our manual methodology to all 1,887 CoCo-flagged extensions through parallel execution with structured exploitability criteria. Comparing LLM classifications against manual ground truth (239 extensions) reveals 81.6% agreement but a 43.7% false negative rate where genuine vulnerabilities are incorrectly classified as benign. Analysis of disagreement patterns identifies systematic limitations in cross-file dependency tracking (31.6% of failures), threat model reasoning (26.3%), and handling obfuscated code (23.7%). These findings demonstrate that LLMs effectively filter obvious false positives (96.1% specificity) but cannot serve as standalone validation authorities due to systematic limitations in multi-step data flow analysis.

RQ3: Does multi-tool agreement between CoCo and DoubleX provide higher precision than individual tool detections?

We analyze the 74 extensions flagged by both CoCo and DoubleX on the 2025 Chrome Web Store dataset. Manual validation reveals 89.2% precision (66 true positives, 8 false positives) for the overlap, representing a $2.5\times$ improvement over CoCo’s overall 36.4% precision and $1.6\times$ improvement over DoubleX’s 56.7% precision. LLM validation performance also improves on this subset, achieving 87.8% agreement and 86.4% sensitivity compared to 81.6% and 55.2% on the overall dataset. These findings demonstrate that multi-tool agreement filters tool-specific false positives and identifies clearer exploitation patterns suitable for prioritized security reviews.

Contributions

1. **First large-scale precision assessment of CoCo.** We applied CoCo to all 197,088 extensions in the Chrome Web Store and conducted manual security analysis on 239 systematically sampled detections, establishing 36.4% precision (90% CI: 32–42%) (??). This represents the first statistically valid precision measurement for CoCo on a contemporary dataset, revealing that approximately one-third of CoCo’s detections represent genuine exploitable vulnerabilities under refined threat model criteria.
2. **Refined threat model and validation methodology.** We developed a four-step validation approach with explicit exploitability criteria based on threat model alignment with CoCo [16] and DoubleX [4] (Section 3.2). Through systematic classification of 239 extensions, we identified specific threat model boundaries—treating hardcoded backend infrastructure as trusted and requiring complete storage exploitation chains—that distinguish genuine vulnerabilities from the three dominant false positive patterns accounting for 63.6% of detections.
3. **Characterization of LLM capabilities and limitations for security analysis.** We implemented an LLM-based validation system using Claude Sonnet 4.5 agents that analyzed all 1,887 CoCo-flagged extensions (Section 5.4). Comparison with manual ground truth reveals 81.6% agreement, 96.1% specificity in filtering false positives, but 43.7% false negative rate. Analysis of 38 missed vulnerabilities identifies systematic failure modes: cross-file

dependency tracking (31.6%), threat model reasoning errors (26.3%), and obfuscated code analysis (23.7%), establishing fundamental limitations of LLM-based vulnerability validation.

4. **Multi-tool agreement as actionable precision indicator.** Manual analysis of 74 extensions flagged by both CoCo and DoubleX achieved 89.2% precision—2.5× higher than CoCo’s overall 36.4% (??). This demonstrates that multi-tool agreement provides an actionable triage strategy for security teams: extensions detected by multiple independent tools represent high-confidence vulnerabilities suitable for prioritized remediation, while tool-specific detections require more careful manual evaluation.

Thesis Structure The remainder of this thesis is structured as follows. Chapter 2 reviews Chrome extension architecture, CoCo’s vulnerability detection approach, and related work on static analysis for browser extensions. Chapter 3 presents our refined threat model, classification criteria, and four-step validation methodology applied in both manual and LLM-based analysis. Chapter 4 describes CoCo’s large-scale deployment modifications, mass processing implementation, and the LLM-based validation workflow including methodology file encoding and parallel agent execution. Chapter 5 presents findings addressing both research questions: CoCo’s 36.4% precision from manual analysis (RQ1), LLM validation achieving 81.6% agreement with systematic limitations (RQ2), and multi-tool agreement yielding 89.2% precision. Chapter 6 concludes with implications for automated security analysis and directions for future work.

Chapter 2

Background

Browser extensions enhance web functionality but operate with privileged access to sensitive browser capabilities. The Chrome Web Store ecosystem has grown substantially, with around 200 new extensions published daily, accumulating over 130,000 extensions in the last two years [6]. Unlike ordinary web pages constrained by the same-origin policy, extensions can invoke privileged operations including cross-origin network requests, cookie manipulation, and persistent storage access. This makes extensions attractive attack targets when vulnerabilities enable privilege escalation from untrusted web content to sensitive browser capabilities.

Recent platform evolution through Manifest V3 [8] introduced architectural improvements targeting code injection vulnerabilities, including stricter Content Security Policy enforcement and restrictions on remotely hosted code. However, a distinct vulnerability class persists: message-passing patterns that create exploitable data flows from adversary-controlled sources to privileged operations. These vulnerabilities arise when architectural design inadvertently connects untrusted contexts to privileged functionality.

This chapter provides the technical foundation for understanding and detecting privilege escalation vulnerabilities. [Section 2.1](#) examines the Chrome extension architecture, permission model ([Section 2.1.1](#)), and isolation mechanisms ([Section 2.1.2](#)) that create security boundaries attackers seek to breach. [Section 2.2](#) presents CoCo, a static analysis tool designed to detect message-passing vulnerabilities by tracking taint flows from adversary-controlled sources to sensitive capability sinks. ?? introduces Claude Code’s agentic capabilities for automating the manual verification process required to validate CoCo’s findings at scale across the Chrome Web Store dataset.

2.1 Chrome Browser Extensions

Chrome browser extensions enhance web browsing functionality by executing privileged operations beyond ordinary web page capabilities. This elevated privilege level introduces security risks when extensions interact with untrusted web content. Understanding extension architecture and communication mechanisms is essential for analyzing privilege escalation vulnerabilities arising from improper data flows at trust boundaries. This section examines the manifest-based permission model defining extension capabilities ([Section 2.1.1](#)), then analyzes the isolation architecture and inter-component communication patterns that create privilege escalation risks ([Section 2.1.2](#)).

2.1.1 Manifest and Permission Model

Every Chrome extension includes a `manifest.json` file that declares the extension’s meta-data, required permissions, and component registration. The manifest specifies which privileged browser APIs the extension can access (e.g., `"permissions": ["cookies", "storage", "<all_urls>"]`), registers background scripts and content scripts (Section 2.1.2) with their execution contexts, and defining which web pages can interact with the extension. Users must grant these permissions at installation time, creating a coarse-grained trust model where extensions receive broad capabilities with limited runtime controls.

The transition from Manifest V2 to Manifest V3 introduced security architectural changes, including the transition from persistent background pages to service workers [11], stricter Content Security Policy (CSP) enforcement to prevent code injection [9], and restrictions on remotely hosted code [10]. While these changes improve code execution defenses, they do not prevent privilege escalation vulnerabilities that occur when extensions create data flow pathways from untrusted sources—such as web pages or malicious extensions—to privileged APIs.

2.1.2 Isolation Architecture and Communication Patterns

Browser extensions consist of JavaScript code executing in different security contexts with an **isolation architecture** that separates untrusted web content from privileged extension functionality, as illustrated in Figure 2.1.

Content scripts execute in web page contexts where they can manipulate the DOM and monitor user interactions, but have restricted API access (only `chrome.runtime.*` messaging APIs). Their primary role is to serve as a sandboxed intermediary between web pages and privileged extension components.

By contrast, **background scripts** (service workers in Manifest V3 [11]) operate in isolated contexts with full access to privileged browser APIs including cross-origin requests, cookie manipulation, and sensitive storage operations.

Extensions communicate across these isolated contexts via message passing mechanisms (`chrome.runtime.sendMessage`, `window.postMessage`). This architecture creates potential privilege escalation vulnerabilities, when attacker-controlled input flows through content scripts to background scripts that invoke sensitive APIs without proper input validation.

Figure 2.1 illustrates the message-passing privilege escalation attack patterns analyzed by CoCo [16]. The figure depicts the browser’s isolation architecture with three distinct layers: the web layer containing both adversary-controlled and benign web pages, the content script layer operating in a shared DOM context with medium privileges, and the background script layer running in an isolated context with high privileges and access to sensitive browser APIs such as `chrome.cookies.*` and `chrome.storage.*`.

In the primary attack flow, an adversary-controlled web page communicates bidirectionally with the victim extension’s content script via `window.postMessage` (Step 1), enabling both malicious data injection and exfiltration of sensitive information. The content script exchanges messages with the background script through `chrome.runtime.*` APIs (Step 2), forwarding tainted data to privileged contexts. The background script invokes sensitive browser APIs (Step 3) using the attacker’s input. Finally (Step 4), the background script can send privileged data to attacker-controlled servers via `fetch()`, completing the attack loop. The figure also illustrates an alternative attack vector where a malicious extension communicates directly with the vulnerable extension’s background script via `chrome.runtime.onMessageExternal`, bypassing the web page intermediary entirely. All communication channels are bidirectional, allowing attackers

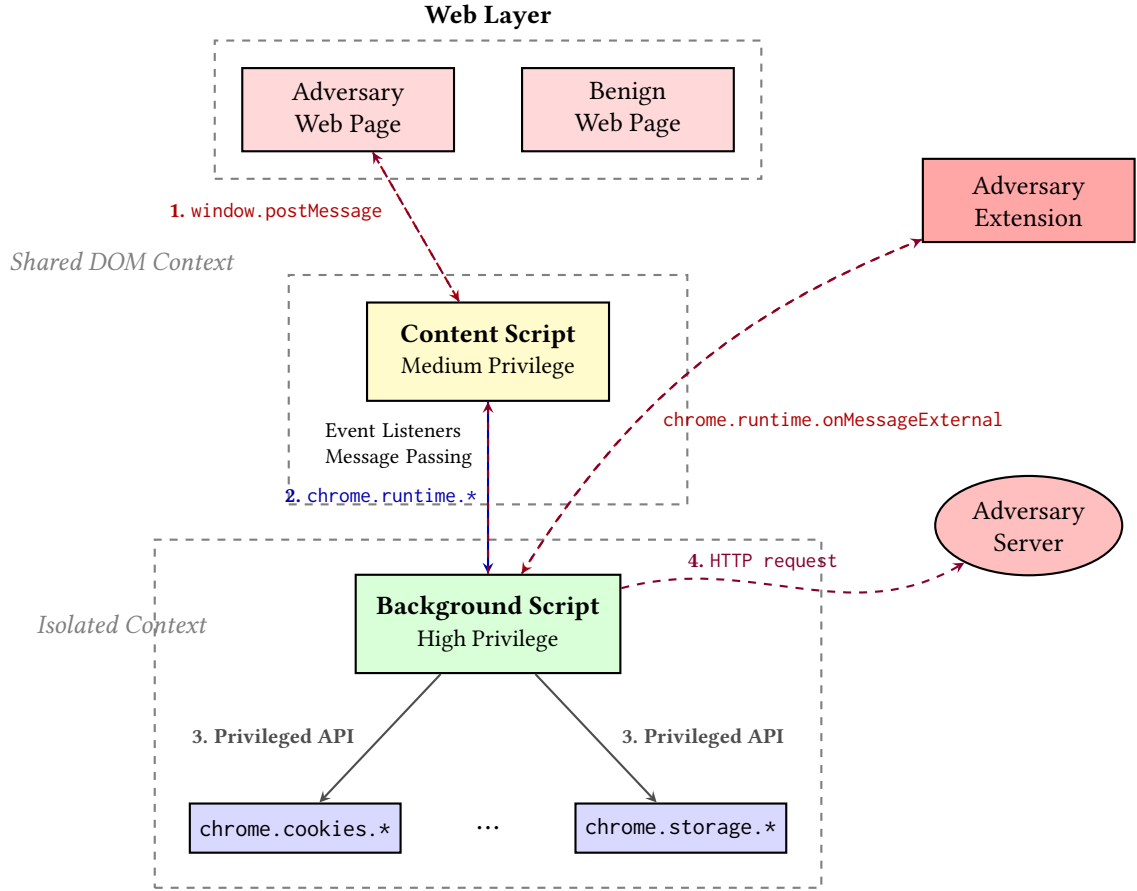


Figure 2.1: Browser extension message-passing privilege escalation attack flow showing bidirectional communication channels and the APIs used at each step.

to both inject malicious commands and receive exfiltrated sensitive data. These attack patterns enable various exploitation scenarios including unauthorized cookie exfiltration, cross-origin data theft, and arbitrary privileged API invocations that would be impossible from the web page or unprivileged extension alone.

2.2 CoCo Vulnerability Detection

CoCo [16] is a static analysis framework that detects message-passing vulnerabilities in browser extensions through coverage-guided, concurrent abstract interpretation. Unlike prior static analysis tools that struggle with dynamic JavaScript features, CoCo uses abstract interpretation on a graph-based abstract domain (Object Dependence Graph) to handle dynamic property access, bracket notation, and asynchronous callbacks. Building on the extension architecture presented in Section 2.1, CoCo specifically targets the privilege escalation attack patterns illustrated in Figure 2.1, tracking taint flows from adversary-controlled sources through message-passing channels to privileged API sinks. This section presents CoCo’s core approach: its graph-based analysis pipeline (Section 2.2.1), concurrent execution model (Section 2.2.1), threat model and vulnerability taxonomy (Section 2.2.2), and analysis output format (Section 2.2.3).

2.2.1 Analysis Pipeline

CoCo’s analysis pipeline operates in three sequential phases: preprocessing and graph construction, concurrent abstract interpretation with taint tracking, and vulnerability detection. This multi-phase approach systematically transforms extension source code into an analyzable representation, tracks how data flows through the extension’s components, and identifies exploitable vulnerability patterns.

Preprocessing and Graph Construction. The analysis begins with a preprocessing phase that prepares the extension’s source code for static analysis. CoCo first examines the extension’s `manifest.json` file to build a comprehensive understanding of the extension’s structure and security boundaries. This manifest analysis serves two purposes: identifying all JavaScript components that must be analyzed (including background scripts, content scripts, and HTML-embedded scripts declared in the manifest) and examining permission declarations.

After identifying the analysis scope, CoCo parses each JavaScript source file using the Esprima parser, which converts the raw JavaScript code into Abstract Syntax Trees (ASTs). These ASTs provide a structured, hierarchical representation of the code’s syntax but do not yet capture the semantic relationships between different code elements. To enable data flow analysis, CoCo transforms these ASTs into an Object Dependence Graph (ODG)—a graph-based abstract domain where nodes represent JavaScript objects, functions, and variables, while edges represent dependencies and data flow relationships. This graph representation is particularly powerful for handling dynamic JavaScript features such as property access via bracket notation (e.g., `obj[dynamicKey]`), where the accessed property cannot be determined until runtime. By maintaining a graph of all potential object properties and their relationships, CoCo can conservatively track how data might flow through dynamically-accessed properties without requiring concrete execution.

Concurrent Taint Analysis. CoCo’s distinguishing feature is its concurrent, coverage-guided abstract interpretation—a novel approach that addresses two fundamental challenges in static analysis of browser extensions: scalability in the face of complex control flow, and comprehensive code coverage despite asynchronous execution patterns. The “two Cos” in CoCo’s name reflect these dual innovations: Concurrent execution and Coverage-guided scheduling.

The concurrent execution model works by parallelizing the abstract interpretation process. When CoCo encounters points where execution paths diverge—such as conditional branches (`if/else` statements), asynchronous event callbacks (e.g., `setTimeout`, `onMessage.addListener`), or message-passing between content and background scripts—it creates separate analysis threads to explore each path independently. This is particularly crucial for browser extensions because their message-passing architecture creates natural concurrency points where content scripts and background scripts operate asynchronously. The coverage-guided scheduling mechanism manages these concurrent threads using a dynamic priority system. CoCo assigns each thread a priority score based on how much new code it has recently covered, its branch depth (favoring higher-level code over deeply nested functions), and how long it has been since the thread last executed (to prevent starvation). Threads that consistently discover new code paths receive more execution time, while threads that repeatedly analyze the same code are preempted to allow other threads to run. This scheduling strategy ensures that CoCo prioritizes the high-level message-passing logic—where vulnerabilities typically manifest—over low-level code.

During analysis, CoCo propagates taint labels from adversary-controlled sources to track how potentially malicious data flows through the extension. The framework marks data as/ tainted

when it originates from sources that an adversary can control, such as message listener parameters (`chrome.runtime.onMessage`, `chrome.runtime.onMessageExternal`), window message events (`window.addEventListener('message', ...)`), or DOM events dispatched by web page content. CoCo then tracks how this tainted data flows through variable assignments, function calls, object property accesses, and message-passing APIs. Critically, CoCo models the semantics of Chrome’s message-passing APIs to understand how taint propagates across component boundaries: when a content script sends a message via `chrome.runtime.sendMessage(data)`, CoCo tracks that the taint associated with data flows to the message parameter of any registered `chrome.runtime.onMessage` listener in the background script. The framework maintains event queues to model asynchronous callback execution and simulates the extension’s event loop to ensure that taint propagation correctly captures the temporal ordering of events.

Vulnerability Detection. CoCo detects vulnerabilities by identifying feasible paths from adversary-controlled sources to sensitive API sinks. This detection operates on two complementary dimensions: control-flow reachability and data-flow taint propagation.

The control-flow reachability analysis determines whether an adversary can actually trigger execution of the sensitive API. CoCo constructs a control-flow graph showing all possible execution paths through the extension’s code and checks whether there exists a path from an adversary-controllable entry point (such as a message listener or DOM event handler) to the sensitive API call. This ensures that CoCo only reports vulnerabilities that an adversary can actually trigger, filtering out API usages that occur in code paths unreachable from adversary-controlled inputs.

The data-flow taint propagation analysis determines whether adversary-controlled data can influence the parameters passed to sensitive APIs. CoCo tracks taint labels through the Object Dependence Graph, following data as it flows through variable assignments, function parameters, return values, and object properties. When CoCo identifies that a tainted value reaches a sensitive API parameter—for example, if the URL parameter of `chrome.tabs.executeScript` is tainted by a message received from a web page.

Finally, CoCo validates that detected vulnerabilities are exploitable under the permission model, by checking the extension’s manifest permissions. Even if an adversary can trigger a sensitive API call with controlled data, the vulnerability is only exploitable if the extension has declared the necessary permissions in its manifest. For instance, a data flow to `chrome.cookies.getAll` only constitutes an exploitable vulnerability if the manifest includes the “cookies” permission.

2.2.2 Threat Model and Vulnerability Types

CoCo [16] targets privilege escalation vulnerabilities where adversaries manipulate victim browser extensions to invoke privileged browser APIs that the adversary could not normally access directly. The threat model assumes two parties: an adversary who operates as an external actor, and a victim extension that contains vulnerable message-passing code. Figure 2.1 illustrates these attack patterns and the privilege escalation flow from adversary-controlled sources to sensitive API sinks.

The adversary can manifest in two forms. In the first scenario, the adversary controls a malicious web page that the user visits while the vulnerable extension is active. This web page can send crafted messages to the extension’s content script via `window.postMessage`, trigger custom DOM events that the content script monitors via `addEventListener`, or exploit any other communication channel that the content script exposes to web page content. In the second scenario, the adversary operates a malicious extension installed alongside the victim extension. If

the victim extension implements `chrome.runtime.onMessageExternal` listeners and does not restrict inter-extension communication via the `externally_connectable` manifest field (which is the default behavior [7]), any malicious extension can send messages directly to the victim's background script via `chrome.runtime.sendMessage` with the victim's extension ID, bypassing the content script intermediary entirely. Both scenarios share the common characteristic that the adversary can supply arbitrary input to the victim extension through standardized browser APIs but cannot directly invoke privileged browser APIs due to permission restrictions—the adversary must induce the victim extension to invoke those APIs on the adversary's behalf.

CoCo detects four categories of privilege escalation vulnerabilities based on the types of sensitive browser APIs that adversaries can abuse, as shown in Table 2.1. These categories represent different classes of harm that attackers can achieve by manipulating vulnerable extensions.

Code Execution vulnerabilities allow adversaries to execute arbitrary JavaScript code under the extension's privilege level. When an adversary can control the parameters of APIs such as `eval`, `tabs.executeScript`, `setTimeout`, or `setInterval`, they can inject malicious code that runs with the extension's elevated permissions. This is particularly dangerous because it effectively grants the adversary all capabilities that the extension possesses, transforming a limited message-passing interface into complete control over the extension's execution environment.

Privileged AJAX Requests enable adversaries to send cross-origin HTTP requests that bypass the same-origin policy enforced on ordinary web pages. Background scripts can send requests to any domain via `fetch` or `XMLHttpRequest` without CORS restrictions, subject only to the host permissions declared in the extension's manifest. If an adversary can control the URL parameter of these APIs, they can exfiltrate data to attacker-controlled servers, probe internal network services not accessible from the web, or relay authenticated requests to third-party services using the user's cookies. Note that CoCo does not consider AJAX requests from content scripts as privileged because content scripts are subject to the same same-origin policy as web pages.

File Downloads permit adversaries to trigger unauthorized file downloads to the user's device through the `chrome.downloads.download` API. An adversary who controls the URL parameter can cause the extension to download malware, phishing pages for offline viewing, or other malicious content. Depending on the browser's download settings and the file type, these downloaded files may execute automatically or deceive users into opening them.

Data Exfiltration through Privileged Access enables adversaries to read or write sensitive browser data that is normally protected from web content. This category encompasses several API families: cookie APIs (`chrome.cookies.*`) that allow reading session tokens and authentication credentials; history and bookmark APIs (`chrome.history.*`, `chrome.bookmarks.*`) that expose users' browsing patterns for tracking or constructing targeted phishing attacks; and browser management APIs (`chrome.topSites.*`, `chrome.management.*`) that reveal information about installed extensions and frequently visited sites.

Privileged Storage Access grants adversaries unauthorized access to the extension's own persistent storage mechanisms. This includes `chrome.storage.local` and `chrome.storage.sync` for extension-specific data, as well as `localStorage` for simpler key-value storage. By manipulating stored data, adversaries can alter the extension's persistent state, potentially corrupting configuration settings, injecting malicious data that will be processed on future runs, or exfiltrating sensitive information that the extension has stored.

Table 2.1: A comprehensive list of sensitive sink APIs modeled by CoCo (updated from Table 1)

Vulnerability Type	Sensitive APIs
Code execution	eval, Function, tabs.executeScript, setTimeout, setInterval
Privileged AJAX requests	ajax, fetch, get, post, XMLHttpRequest().open
File downloads	downloads.download, downloads.removeFile, downloads.erase
Data exfiltration - Privileged Access	cookies.get, cookies.getAll, cookies.set, cookies.remove, bookmarks.getTree, bookmarks.create, bookmarks.remove, bookmarks.search, history.search, history.getVisits
Privileged Storage Access	topSites.get, management.getAll, management.getSelf, management.setEnabled, storage.local.get, storage.sync.get, storage.local.set, storage.sync.set, storage.local.clear, storage.sync.clear, localStorage.getItem, localStorage.setItem, localStorage.removeItem, localStorage.clear
Tab & Browser Manipulation	tabs.create, tabs.update, browsingData.remove

2.2.3 Analysis Output

CoCo generates analysis results in an `opgen.generated_files/` directory created within each analyzed extension’s folder. This output directory contains three types of files: vulnerability detection reports (`used_time.txt`), instrumented source code for manual inspection (`bg.js`, `cs_*.js`), and a copy of the extension’s `manifest.json` for permission verification.

Vulnerability Detection Format The primary output file, `used_time.txt`, records vulnerability detections with structured taint flow information. Each detection follows a standardized format that facilitates automated parsing and classification. The detection begins with a header line identifying the vulnerable extension and sink type, followed by a source-to-sink summary and a detailed line-by-line taint flow path.

Figure 2.2 shows a real detection from extension `abkbmnbcmfehcbfjkpagdmloiondlbne` where attacker-controlled data from a malicious webpage flows to `chrome.tabs.executeScript`, representing a code execution vulnerability. The source type `cs_window_eventListener_message` indicates the content script receives attacker data via `window.postMessage`, while the sink type `chrome_tabs_executeScript_sink` specifies arbitrary code execution. The taint flow path shows how `event.data` propagates through message passing to the background script, where `request.windowTitle` is directly concatenated into the `code` parameter of `executeScript`, enabling arbitrary JavaScript injection into the victim’s tab.

Automated Classification These structured source and sink type identifiers enable classification of vulnerabilities into the categories defined in Table 2.1. For example, sink types matching `eval_sink`, `tabs.executeScript_sink`, `setTimeout_sink`, or `setInterval_sink` indicate Code

```

1 tainted detected!~~~ in extension: ../abkbmnbcmfehcbfjkpagdmloiondlbne
2 with chrome_tabs_executeScript_sink
3 from cs_window_eventListener_message to chrome_tabs_executeScript_sink
4 =====
5 $FilePath$../opgen.generated.files/cs_0.js
6 Line 483 window.addEventListener("message", function(event) {
7   event
8   $FilePath$../opgen.generated.files/cs_0.js
9   Line 487 if (event.data.type && (event.data.type == "LV-PCIDSS-TRIGGER")) {
10    event.data
11    $FilePath$../opgen.generated.files/bg.js
12    Line 977 if (request.event == "PAUSE" && ...) {
13      request.windowTitle
14      $FilePath$../opgen.generated.files/bg.js
15      Line 979 chrome.tabs.executeScript(tab.id,
16        {code:"document.title = '' + curTitle + request.windowTitle + ''});
17
18 finish within 0.50 seconds

```

Listing 2.1: CoCo vulnerability detection output from extension
abkbmnbcmfehcbfjkpagdmloiondlbne

Figure 2.2: CoCo vulnerability detection output showing webpage-to-extension code execution vulnerability. Attacker-controlled event.data from postMessage flows through content script to background script, reaching chrome.tabs.executeScript via string concatenation in request.windowTitle. Tainted data flow highlighted in red.

Execution vulnerabilities, while cookies*_sink or history*_sink types indicate Data Exfiltration. This automated classification is implemented in the categorization script described in [Section 4.4](#).

Instrumented Source Files CoCo produces instrumented versions of the extension’s source code in files named bg.js (background scripts) and cs_*.js (content scripts). These files combine CoCo’s taint-tracking instrumentation code with the extension’s original source code, separated by // original [file-path] markers. Line numbers in vulnerability reports may reference either CoCo’s instrumentation code or the extension’s original code, a distinction critical for automated validation workflows ([Section 3.2.4](#)).

Analysis Status Indicators Analysis completion status is indicated by specific marker strings in used_time.txt. A successful analysis writes finish within X seconds, while a timeout writes timeout after Y seconds. These status indicators enable the categorization script ([Section 4.4](#)) to classify extensions into vulnerable, timeout, parser error, and successful categories for understanding CoCo’s coverage across the dataset.

2.3 Related Static Analysis Tools

Beyond CoCo, several static analysis tools target browser extension security through different technical approaches and threat models. We review DoubleX and CodeX, two complementary tools that informed our threat model refinements, particularly regarding storage API treatment and contextual flow analysis. Extensions flagged by multiple tools represent higher-confidence detections that satisfy different criteria, potentially indicating more straightforward vulnerability patterns. Our analysis compares CoCo and DoubleX detections on the 2025 Chrome Web Store

dataset to assess how threat model variations affect precision measurements and whether multi-tool agreement correlates with genuine exploitability under our refined threat model.

2.3.1 DoubleX: Hybrid Taint Tracking

DoubleX [4] represents an alternative static analysis approach for detecting privilege escalation vulnerabilities in browser extensions. While CoCo and DoubleX share the fundamental goal of identifying flows from attacker-controlled sources to privileged API sinks, they differ in their technical approaches and vulnerability criteria.

DoubleX employs hybrid analysis combining lightweight static taint tracking with selective dynamic execution for validating detected flows, prioritizing scalability through syntactic pattern matching over CoCo’s deeper semantic modeling. This technical difference manifests in their sink specifications: CoCo models 57 distinct sink types across categories including storage operations, network requests, and tab manipulation (Table 2.1), while DoubleX focuses on a more selective set of high-impact sinks emphasizing code execution and direct data exfiltration paths.

A critical distinction lies in storage API treatment: CoCo conservatively flags all flows to storage write operations (`chrome.storage.set`, `localStorage.setItem`) as vulnerabilities regardless of subsequent use. DoubleX requires demonstrating complete exploitation chains from storage writes through subsequent reads to attacker-accessible outputs, applying stricter exploitability criteria that reduce false positive rates. This distinction directly informs our threat model (Section 3.1.3), where we adopt DoubleX’s approach of requiring demonstrated retrieval paths to attacker-accessible outputs.

2.3.2 CodeX: Contextual Flow Tracking

CodeX [2] introduces contextual flow tracking for detecting privacy violations in browser extensions, tracking flows from sensitive data sources (cookies, browsing history, bookmarks, search terms) to network destinations. CodeX extends traditional taint tracking with contextual information about destination URLs to distinguish between flows to developer-controlled infrastructure versus attacker-accessible endpoints.

CodeX’s key contribution lies in its treatment of hardcoded backend URLs. Unlike vulnerability-focused tools that flag all sensitive data flows to network sinks, CodeX classifies flows targeting hardcoded developer URLs as distinct from flows to attacker-controlled destinations, recognizing that developers implicitly trust their own infrastructure. This contextual flow tracking directly informs our threat model (Section 3.1.3), where we exclude flows involving developer-controlled URLs from exploitability assessment since we cannot assess backend security through extension code analysis alone.

CodeX’s approach demonstrates that contextual information—particularly destination URLs—enables more precise classification of detected flows. Our threat model adopts this principle by distinguishing between three destination categories: hardcoded developer backends (trusted), attacker-controlled URL parameters (untrusted), and attacker-accessible outputs like `postMessage` (untrusted). This refinement focuses vulnerability analysis on extension code issues developers can remediate, rather than backend infrastructure security outside the scope of static code analysis.

Chapter 3

Methodology

This chapter presents our systematic approach to validating CoCo’s vulnerability detections. We establish our threat model (Section 3.1), defining adversary capabilities, taint sources and sinks, and exploitable impact criteria that focus on direct exploitability within extension code. We then present our validation methodology (Section 3.2), a four-step process applied through manual ground truth analysis and scalable LLM-based validation.

3.1 Threat Model

We validate CoCo’s vulnerability detections under a threat model that requires flows to demonstrate direct exploitability within extension code. Our approach analyzes CoCo’s taint tracking results [16] through refined exploitability criteria informed by DoubleX [4] and CodeX [2]. We focus on privilege escalation attacks where adversaries leverage extension code vulnerabilities to access privileged browser APIs or sensitive user data.

Our threat model establishes: adversary capabilities and attack vectors (Section 3.1.1), taint sources and privileged sinks tracked by CoCo (Section 3.1.2), and exploitability criteria defining when detected flows constitute genuine vulnerabilities (Section 3.1.3). Two principles distinguish our model from CoCo’s conservative approach: we treat hardcoded developer infrastructure as trusted [2], and we require complete exploitation chains for storage flows [4]. These principles are integrated into our exploitability definition rather than applied as post-hoc filters.

3.1.1 Adversary Capabilities

We consider two adversary types that can exploit extension message-passing interfaces. Neither can directly invoke privileged browser capabilities and must manipulate the victim extension to perform privileged operations on their behalf.

Malicious Webpage Adversary A malicious webpage visited by the user can interact with extension content scripts through `window.postMessage`, custom DOM events, or manipulated DOM elements. The webpage operates with standard web origin restrictions but can leverage the extension’s privileged capabilities by injecting malicious data through these communication channels.

Malicious Extension Adversary A malicious installed extension can send messages to the victim extension’s background script via `chrome.runtime.onMessageExternal` when the victim ex-

tension does not restrict cross-extension communication through the `externally_connectable` manifest field [7]. This adversary bypasses content script intermediaries entirely.

3.1.2 Privileged Sinks

We analyze taint flows reaching privileged operations that adversaries cannot directly invoke but can exploit through vulnerable extensions. CoCo tracks flows from adversary-controlled entry points (defined in [Section 3.1.1](#)) to privileged sinks unavailable to ordinary web content [16]. Following CodeX’s contextual flow tracking approach [2], we assess exploitability based on both flow presence and contextual information including destination URLs and complete exploitation chains. CoCo monitors flows to the following privileged sink categories:

- **Code execution:** `eval`, `setTimeout`, `setInterval`, `Function()`, `chrome.tabs.executeScript`
- **Network requests:** `fetch`, `XMLHttpRequest`, framework-specific HTTP clients (`jQuery.ajax`, `Axios`, `Ky`)
- **Browser manipulation:** `chrome.tabs.*`, `chrome.webRequest.*`
- **Message passing:** `postMessage`, `sendResponse`
- **Storage operations:** `chrome.storage.*.set`, `localStorage.setItem`

These sinks enable privilege escalation when reached by attacker-controlled data, but exploitability depends on contextual factors examined in the following subsection.

3.1.3 Exploitable Impact Criteria

A detected taint flow constitutes an exploitable vulnerability only when it enables privilege escalation through extension code alone—accessing browser APIs or user data unavailable to the adversary directly. Our exploitability assessment incorporates two key principles that distinguish exploitable flows from benign patterns:

Principle 1: Hardcoded Developer Infrastructure is Trusted Following CodeX [2], we treat flows involving hardcoded developer-controlled backend URLs as non-exploitable. Extension developers implicitly trust their own infrastructure, making backend security outside the scope of extension code vulnerability analysis. We cannot assess whether developer backends perform adequate validation or whether backend compromise risks exist through static analysis of extension code alone.

A backend URL is developer-controlled when hardcoded as string literals, configured in bundled JSON files or other files in the extension. Flows to or from such URLs—whether carrying attacker data to backends, retrieving backend data to dangerous sinks, or sending storage data to backends—represent developer infrastructure trust decisions rather than extension code vulnerabilities.

Principle 2: Storage Flows Require Complete Exploitation Chains Following DoubleX [4], we require storage flows to demonstrate complete exploitation chains. While CoCo conservatively flags all flows reaching `storage.set` as vulnerable, we recognize that storage write operations alone do not constitute exploitable vulnerabilities. Exploitability requires demonstrating that stored data flows back to attacker-accessible outputs, enabling the attacker to observe the result of their privileged storage access.

Exploitable Impact Categories Under these principles, we classify flows as exploitable when they achieve privilege escalation through one of the following patterns:

- **Arbitrary code execution:** Attacker data \rightarrow `eval`, `setTimeout`, `setInterval`, `Function()`, or `chrome.tabs.executeScript`
- **Cross-origin data exfiltration:** Attacker data \rightarrow network requests to attacker-controlled URL parameters (excluding hardcoded developer backends)
- **Sensitive data exfiltration:** User data (cookies, history, bookmarks) \rightarrow attacker-accessible outputs (`postMessage`, `sendResponse`, network requests to attacker-controlled URLs)
- **Storage data exfiltration:** Pre-existing storage data \rightarrow `storage.get` \rightarrow attacker-accessible outputs, exposing authentication tokens, user preferences, or tracking identifiers [13]
- **Storage oracle privilege escalation:** Attacker data \rightarrow `storage.set` \rightarrow `storage.get` \rightarrow attacker-accessible outputs. Even when attackers store their own data, retrieving it from storage represents privilege escalation by gaining access to storage APIs normally unavailable to web content.

Flows lacking attacker-accessible outputs (e.g., `storage.set` without subsequent `get` \rightarrow output) or targeting developer infrastructure (e.g., `fetch("https://hardcoded-backend.com")`) do not achieve exploitable impact under our threat model.

3.1.4 Domain Restrictions Are Not Security Boundaries

Following CoCo and DoubleX [16, 4], we do not treat domain restrictions as security boundaries when evaluating message passing exploitability. We consider any website where the extension runs as a potential attack surface, regardless of `content_scripts` match patterns or `host_permissions` restrictions in the manifest.

We classify extensions as vulnerable (true positive) even when restricted to a single specific domain, focusing on whether a data flow could be exploited if the adversary controls that webpage. This approach aligns with CoCo and DoubleX, which both report as vulnerable extensions limited to specific domains [16, 4]. Verifying third-party website security posture falls outside extension security analysis scope.

Similarly, extensions with `chrome.runtime.onMessageExternal` handlers are considered vulnerable to malicious extensions, regardless of `externally_connectable` manifest restrictions, as any installed extension can send messages to these handlers by default [7].

3.2 Vulnerability Validation Methodology

Building on the threat model defined in Section 3.1, we developed a systematic validation methodology that evaluates each CoCo detection against explicit exploitability criteria. We focus our analysis on CoCo’s detected flows to specifically measure its precision—whether flagged flows constitute genuine exploitable vulnerabilities under our threat model requiring direct exploitability within extension code.

Our validation follows a four-step process (Section 3.2.1) that traces data flows, verifies permissions, assesses exploitability, and applies binary classification criteria (Section 3.2.2). We apply this approach through two complementary strategies: manual security analysis on statistically

sampled datasets establishes ground truth (Section 3.2.3), while LLM-based automated analysis enables validation across all 1,887 CoCo-flagged extensions (Section 3.2.4).

3.2.1 Four-Step Validation Approach

Our validation follows a systematic four-step process for each CoCo-flagged extension:

1. **Extract CoCo Detection Information:** Parse CoCo’s output (`used_time.txt`) to identify source→sink flows, flagged line numbers, and involved files.
2. **Trace Data Flow and Verify Permissions:** Reconstruct the complete vulnerability path through extension code and verify that required permissions exist in `manifest.json`.
3. **Assess Exploitability:** Evaluate whether an external attacker can trigger the flow, control data values, and achieve exploitable impact.
4. **Classify and Document:** Apply binary classification criteria and document the rationale.

3.2.2 Classification Criteria

We apply binary classification to each CoCo detection based on whether it represents an exploitable vulnerability under our refined threat model. Classification requires evaluating all conditions in conjunction—a detection constitutes a true vulnerability only when every criterion is satisfied, while any single failing criterion indicates a false positive.

True Positive: Exploitable Vulnerability We classify a finding as **True Positive** when ALL five conditions hold:

1. **Real code flow:** The data flow exists in actual extension code (not CoCo instrumentation artifacts)
2. **External attacker trigger:** An external attacker can trigger the flow (via malicious webpage, malicious extension, or DOM manipulation)
3. **Manifest permissions:** Required permissions are declared in `manifest.json`
4. **Attacker control:** The attacker can control data values reaching the sink
5. **Exploitable impact:** The flow achieves privilege escalation as defined in Section 3.1.3

False Positive: Benign Flow We classify as **False Positive** when ANY condition fails. Common false positive patterns include:

1. **Framework-only flows:** CoCo instrumentation artifacts without corresponding flows in original code
2. **No external attacker trigger:** User inputs in extension’s own UI, developer-initiated actions, or internal extension logic
3. **Missing required permissions:** Flows to APIs without corresponding manifest permissions

4. **No attacker-controlled data:** Flows involving hardcoded backend URLs or developer-controlled constants
5. **No exploitable impact:** Storage writes without demonstrated retrieval paths to attacker-accessible outputs, or data flows to trusted developer infrastructure

The exploitable impact criteria defined in [Section 3.1](#) guide our classification decisions.

3.2.3 Manual Validation Approach

Manual analysis serves two critical purposes in our evaluation: establishing ground truth for measuring detection precision under our refined threat model, and providing reference classifications for validating LLM-based analysis accuracy. We apply our four-step validation methodology ([Section 3.2.1](#)) through detailed code review of systematically selected extension samples. Manual classifications provide authoritative ground truth because human analysts can apply contextual reasoning about threat model boundaries, trace complex data flows across files, and identify subtle exploitation patterns that require semantic understanding of code intent.

Dataset Selection

We manually analyze three extension sets, each serving distinct validation purposes:

1. **CoCo 2021 Dataset:** All 39 extensions flagged in the original CoCo paper [16], enabling comparison with prior work and validation of our threat model refinements against the baseline dataset.
2. **CoCo 2025 Dataset Sample:** 239 extensions from the 1,887 CoCo-flagged extensions, selected using statistical sampling principles. With a population of 1,887 extensions, a sample of 239 provides 90% confidence that measured values fall within $\pm 5\%$ of true population values. This sample enables statistically valid measurement of both detection precision under our refined threat model and LLM classification accuracy.
3. **CoCo-DoubleX Overlap:** All 74 extensions flagged by both CoCo and DoubleX [1] on the 2025 dataset. This complete overlap analysis enables direct comparison of multi-tool agreement precision against individual tool performance, addressing whether extensions detected by multiple independent static analysis tools represent higher-confidence vulnerabilities.

3.2.4 LLM-Based Validation Approach

To assess whether automated techniques can replicate human security analysis at scale, we employ LLM-based validation that applies the four-step validation methodology ([Section 3.2.1](#)) to all 1,887 CoCo-flagged extensions from the 2025 dataset. We use Claude Sonnet 4.5 through the Claude Code agent framework [3], implementing parallel subagent execution where each agent independently analyzes multiple extensions following our validation approach. This comprehensive LLM analysis enables two complementary evaluations: (i) measuring LLM classification reliability by comparing against manual ground truth on the 239-extension sample, and (ii) applying automated validation at scale across the complete dataset to assess the overall distribution of genuine vulnerabilities. Additionally, we apply the LLM validation to the 39 CoCo extensions reported in the original paper and 74 CoCo-DoubleX overlap extensions to evaluate LLM performance across different dataset contexts.

Methodology Encoding

Each subagent receives a structured prompt file (`CoCo_Analysis_Methodology.md`) that encodes the vulnerability validation methodology presented in [Section 3.2](#) into explicit LLM-executable instructions. The file specifies the four-step validation process ([Section 3.2.1](#)), classification criteria ([Section 3.2.2](#)), and threat model principles ([Section 3.1](#)), including three critical rules that address common misclassification patterns: (i) ignore manifest restrictions on message passing, (ii) require complete exploitation chains for storage flows, and (iii) treat hardcoded backend URLs as trusted infrastructure. The complete methodology file specification is detailed in [Section 4.5.1](#).

Parallel Analysis Strategy

We employ an iterative batch architecture with 10 independent subagents analyzing disjoint subsets of 5 extensions each (50 extensions per batch). This configuration balances dataset scale, analysis complexity, and resource limitations (200,000 token context window per subagent, 10 concurrent subagent limit per Claude Code’s documented constraints [?]). Implementation details, including assignment scripts and batch execution workflow, are presented in [Section 4.5](#).

Chapter 4

Implementation

This chapter describes the practical implementation of our analysis pipeline for the 197,088-extension dataset. We present: (i) CoCo execution at scale (Section 4.1), including infrastructure setup and invocation details, (ii) the custom mass processor script (Section 4.3) that orchestrates parallel CoCo execution with reliable timeout enforcement, (iii) the results categorization script (Section 4.4), which classifies extensions into vulnerable, timeout, parser error, and successful categories and (iv) LLM-based analysis implementation (Section 4.5).

4.1 CoCo Execution at Scale

We employ CoCo [16], a coverage-guided concurrent abstract interpretation tool, to identify potential security vulnerabilities across the Chrome Web Store dataset. As described in Section 2.2.1, CoCo performs static taint analysis to detect five categories of privilege escalation vulnerabilities (Section 2.2.2). However, applying CoCo to a large dataset required several practical adaptations to address timeout handling, parallel processing, and resume processing logic. Initial testing revealed two critical limitations in CoCo’s architecture that prevented large-scale analysis: unreliable timeout enforcement and no checkpoint savings/resume logic during the full analysis of a dataset (Section 4.2).

4.1.1 CoCo Invocation Command

Each worker thread invokes CoCo using the following command structure:

```
1 python3 /path/to/CoCo/generate_opg.py -t chrome_ext -crx
2     --timeout 600 /path/to/extension
```

Listing 4.1: CoCo invocation command structure

The parameters specify:

- `-t chrome_ext`: Selects Chrome extension analysis mode, enabling message-passing analysis and browser API modeling.
- `-crx`: Indicates Chrome extension format (automatically unpacks if necessary).
- `--timeout 600`: Sets the user-specified analysis timeout to 600 seconds (10 minutes). This parameter configures CoCo’s internal signal-based timeout mechanism, though as described in Section 4.2, this mechanism is unreliable and requires external enforcement.

The command is executed via `subprocess.run()` with `capture_output=True` to collect CoCo’s output for vulnerability detection parsing. Our mass processor (Section 4.3) enforces an external timeout by wrapping the CoCo invocation with `subprocess.run(timeout=630)`, adding a 30-second buffer to the user-specified 600-second timeout. When this 630-second limit is exceeded, `subprocess.run()` sends `SIGKILL` to forcefully terminate the entire CoCo process tree, bypassing the internal timeout mechanism’s limitations.

4.2 CoCo Code Problems and Custom Orchestration

Running CoCo at scale across 197,088 extensions revealed critical limitations in CoCo’s codebase that prevented reliable dataset-wide analysis. Two specific code defects necessitated external orchestration: (1) CoCo’s signal-based timeout mechanism fails to interrupt hung worker threads, and (2) CoCo’s built-in `--parallel` option lacks progress persistence across script restarts. These issues forced us to abandon CoCo’s internal parallelization and instead invoke CoCo separately for each individual extension through a custom orchestration script with external timeout enforcement.

4.2.1 CoCo Timeout Mechanism Failure

CoCo implements timeout enforcement using Python’s signal-based timeout mechanism `signal.SIGALRM` (`src/core/timeout.py:15`), which cannot propagate across thread boundaries. When the user specifies `--timeout 600`, CoCo registers a signal handler that should terminate analysis after 600 seconds. However, this signal-based approach fails in CoCo’s concurrent architecture due to a fundamental limitation of UNIX signals: **signal handlers only execute in the main thread and cannot interrupt blocking system calls in worker threads** [12].

When CoCo registers `signal.SIGALRM` in the main thread (`src/core/timeout.py:14`), the resulting `TimeoutError` exception is raised exclusively in the main thread’s execution context and cannot interrupt worker threads executing in separate call stacks. Worker threads invoking `subprocess.communicate()` (`src/core/esprima.py:15`) enter blocking system calls (kernel-level `read()` operations on subprocess pipes) that are not interruptible by signals delivered to other threads. Python’s Global Interpreter Lock ensures that only the main thread checks for pending signals; worker threads suspended in kernel I/O never return to Python’s signal-checking code path. The main thread’s timeout exception handler attempts cleanup by calling `thread.info.stop()` (`src/core/opgen.py:229`), but this merely sets flags that blocked worker threads never observe while waiting for subprocess I/O.

We observed instances where single-extension CoCo processes continued running for over 24 hours despite the user-specified 600-second timeout, consuming server resources and preventing progress across the dataset.

4.2.2 CoCo Parallel Mode Lacks Progress Persistence

CoCo provides a `--parallel` option (`src/core/opgen.py:374`) for processing multiple extensions concurrently. This implementation spawns separate CoCo processes via shell screen sessions, with each process analyzing a single extension. However, CoCo’s parallel mode lacks three critical features for production dataset analysis:

1. **No progress state persistence:** CoCo maintains no record of which extensions completed successfully. When the script terminates (due to server maintenance, resource exhaustion,

or manual interruption), restarting analysis processes all extensions from the beginning, wasting potentially days of computation.

2. **No centralized error handling:** Each screen session runs independently without coordinated timeout enforcement. When individual CoCo processes hang due to the timeout bug (Section 4.2.1), the parallel orchestrator cannot detect or terminate them externally.
3. **No dynamic resource adjustment:** The parallelism level is fixed at script start. Analyzing 197,088 extensions across multiple days with varying server resource availability requires dynamically adjusting worker counts between runs, which CoCo’s parallel mode does not support.

These limitations make CoCo’s `--parallel` mode unsuitable for large-scale analysis requiring incremental progress across multiple script executions. To address these issues, we implemented a custom orchestration script described in Section 4.3.

4.3 Mass Processor Implementation

We implemented a custom orchestration script (`server_mass_processor.py`) to enable large-scale CoCo analysis across the 197,088-extension dataset. The script addresses the critical limitations identified in Section 4.2: reliable timeout enforcement, persistent progress tracking across multiple runs, and dynamic parallelization configuration scaling with available hardware. In order to analyze the full dataset in a timely manner, we created a thread manager script that would start multiple workers in parallel, which will start themselves CoCo on a given extension. The script treats each CoCo invocation as an isolated subprocess with kernel-level timeout enforcement. We use Python’s `ThreadPoolExecutor` to spawn multiple worker threads based on the resources and time allocated for the analysis. Each worker analyzes assigned extensions independently by invoking CoCo through `subprocess.run()` with a 630-second timeout: 600s internal CoCo timeout plus a 30 seconds buffer to ensure CoCo properly handles the timeout in case it was invoked. When a process exceeds this limit, the subprocess module sends `SIGKILL` to forcefully terminate the hung CoCo process, circumventing CoCo’s unreliable signal-based timeout mechanism (Section 4.2.1).

The script maintains persistent state to enable resumption across multiple runs. We track completed extensions in `processed_extensions.txt` (one extension ID per line) and aggregate statistics in `progress.json`. On restart, the discovery phase excludes extensions already present in `processed_extensions.txt`, preventing duplicate analysis. This resumption capability proved essential when server maintenance required interrupting analysis mid-batch or when adjusting parallelization parameters between runs to speed up the process or reduce the resource consumption during peak hours. The script maintains persistent state to enable resumption across multiple runs. We track completed extensions and logs across runs, enabling the resumption and scalability of the process.

4.4 Categorization Script for CoCo Report

We implemented a post-processing categorization script (`coco_results_categorization.py`) to systematically classify the 197,088 analyzed extensions based on their CoCo analysis outcomes.

The script serves three purposes: extracting vulnerable extensions for manual and LLM validation (Section 3.2.3, Section 3.2.4), distinguishing genuine failures from timeouts, and computing vulnerability distribution statistics across the dataset (Table 5.3).

The script analyzes CoCo's `used_time.txt` output file for each extension (Section 2.2.3) to detect three outcome categories through regex pattern matching on CoCo's status markers and output patterns. CoCo marks analysis outcomes through specific status indicators and structured output as detailed in Section 2.2.3. The following subsections describe how we detect each category.

Vulnerability Detection

CoCo reports detected vulnerabilities through structured taint flow descriptions in `used_time.txt`, listing source types, sink types, and line numbers (Section 2.2.3). We detect these vulnerability reports through regex pattern matching on the taint flow format, extracting sink types reached by tainted data flows (e.g., `chrome.storage.local.set`, `eval`, `XMLHttpRequest`). The script maintains a global mapping from each sink type to affected extension IDs. This approach prevents double-counting: when a single extension contains multiple taint flows to the same sink, we count it once per sink type rather than once per flow. Vulnerability statistics therefore reflect unique extensions affected, not raw taint flow counts (Table 5.3, Table 5.1).

Timeout Detection

CoCo marks timeout events through specific status indicators in `used_time.txt` (Section 2.2.3): successful analyses write `finish within X seconds`, while timeouts write `timeout after Y seconds`. We detect two timeout scenarios through pattern matching on these markers to assess CoCo's reliability under resource constraints. CoCo-enforced timeouts occur when CoCo's internal signal-based mechanism successfully detects resource exhaustion and gracefully terminates analysis, identified by the `timeout after` marker. Mass-analyzer-killed timeouts occur when CoCo's mechanism fails and the mass processor forcefully terminates the hung process via `SIGKILL`, identified when `used_time.txt` contains `analysis starts` but lacks both `finish within` and the explicit `timeout` marker. This distinction enables assessment of timeout mechanism reliability across the dataset (Table 5.1).

Error Detection

CoCo writes error messages to `used_time.txt` when analysis fails due to parsing or structural issues (Section 2.2.3). The error messages follow a structured format `Error: [extension-id] [error-description]`, enabling systematic detection through regex pattern matching. We detect three types of parser errors based on identified log patterns from the CoCo source code (`src/core/helpers.py`):

- **Extension generation errors:** in generating extension files – malformed extension structure or unsupported APIs preventing initial file processing
- **Esprima parse errors:** unexpected token while parsed with esprima – invalid JavaScript syntax or modern language features beyond the parser's capabilities
- **Graph construction errors:** can not import from string – unresolvable dynamic imports or complex module structures that prevent control flow graph construction

The script records extension IDs and error types in `parser_error_report.json`, maintaining up to five sample error messages per type for diagnostic analysis. Results are presented in [Table 5.1](#).

4.5 LLM-Based Analysis Workflow

This section describes the practical implementation of the LLM-based vulnerability validation pipeline introduced in [Section 3.2.4](#). We detail the assignment script that coordinates parallel analysis, the execution workflow for launching agent batches, and the output structure that enables progress tracking across multiple batch executions.

4.5.1 Methodology Encoding for LLM Analysis

The validation methodology ([Section 3.2.1](#)) and classification criteria ([Section 3.2.2](#)) are encoded in a structured prompt file (`CoCo_Analysis_Methodology.md`) that guides LLM agents through systematic vulnerability assessment. The methodology file implements CoCo-specific analysis requirements:

- **CoCo output parsing:** Instructions for extracting source→sink flows from `used_time.txt`, distinguishing internal trace IDs from actual line numbers (‘‘Line X’’)
- **Instrumented file navigation:** Locating original extension code after the third `// original [file-path]` marker in CoCo-generated files
- **Output format specification:** Structured markdown documentation including binary classification, code paths, and for true positives, proof-of-concept exploits

The methodology encodes common misclassification patterns identified during preliminary validation, providing concrete examples of typical true positive sources (DOM events, `window.postMessage`, external messages) and false positive patterns (extension UI inputs, missing permissions, incomplete storage chains).

4.5.2 Batch Execution Workflow

Each batch analysis follows a standardized execution workflow initiated through direct interaction with Claude Code’s command-line interface. [Figure 4.1](#) illustrates the complete workflow from user command to output generation.

One could provide either a single instruction that specifies the methodology file path, the assignment script command, and the directive to launch 10 subagents. This would be considered a batch and once the agents finish the main agent will inform the user. This can be avoided by also prompting the agent to repeat the previous mentioned steps of getting the new extension ids and starting the subagents, thus leaving the agent in a loop until it runs out of context. In that case the researcher would interfere and start a new agent from scratch to repeat the process. A typical invocation takes the following form:

```
1 read '\CoCo_Analysis_Methodology.md', run the script \get_assignments.py to get the next 5 extensions
   for each of the 10 sub-agents and then proceed to start the agents. Instruct the agents to read
   the methodology file and analyze their next 5 extensions based on it.
```

Upon receiving this instruction, the main Claude Code agent follows the workflow shown in [Figure 4.1](#). The agent reads the methodology file and executes the assignment script to obtain 50

extension IDs partitioned into 10 groups. The agent then spawns 10 independent subagents using Task tool invocations, providing each with an initialization prompt containing the methodology file path and their specific list of 5 extension IDs. Each subagent autonomously processes its assigned extensions following the validation methodology (Section 3.2.1), writing analysis reports to `{extension_id}_analysis.md` files. This produces 50 markdown files per batch.

The main agent monitors subagent completion and reports when all 10 subagents finish. If a subagent exhausts its token quota mid-analysis (consuming its 200,000 token context window), it terminates without producing output files for incomplete extensions. The multi-batch progress tracking mechanism (Section 4.5.3) handles these partial failures automatically in subsequent batch executions.

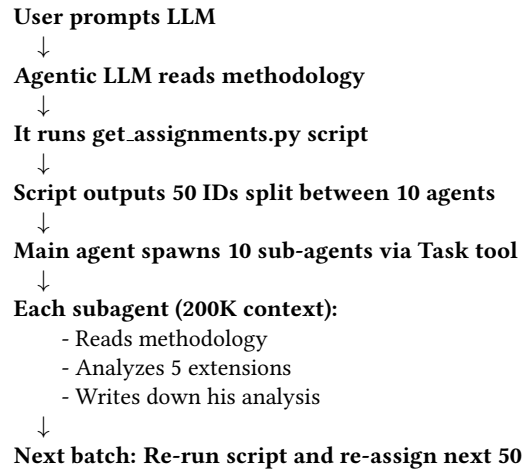


Figure 4.1: Agent batch workflow.

4.5.3 Multi-Batch Progress Tracking

Analyzing the complete dataset of 1,887 extensions requires approximately 38 batch executions ($1,887 \div 50 = 37.74$ batches). We manually initiated each batch by re-issuing the batch execution command after the previous batch completes. We also tested with an initial command that was encouraging the agent to rerun the command until the context was full and it proved to only need human intervention every four to five batches. The assignment script automatically identifies the next 50 unanalyzed extensions, enabling seamless continuation without requiring manual tracking of which extensions have been processed.

The presence of `*_analysis.md` files in the output directory serves as the sole progress state. This stateless design provides robustness against interruptions: if a batch fails partially (some agents complete while others exhaust tokens), the next batch execution automatically identifies and processes the extensions that failed in the previous attempt. No separate progress database or lock files require maintenance. This resumption mechanism proved essential for completing the dataset-wide analysis across multiple sessions.

4.5.4 Agent Runtime Metrics

Analyzing all 1,887 CoCo-flagged extensions required 377 agents ($1,887 \div 5$ extensions per agent). We collected performance logs from 247 agent executions (65.5%), representing 1,235 extensions

analyzed. Table 4.1 presents token consumption and execution time statistics across the distribution. The table reports six statistical measures: minimum (Min) and maximum (Max) values representing the extreme cases, first quartile (Q1, 25th percentile) indicating that 25% of agents consumed fewer resources than this value, third quartile (Q3, 75th percentile) indicating that 75% of agents consumed fewer resources than this value, median representing the midpoint where half of agents consumed more and half consumed less, and mean representing the arithmetic average across all agents.

Table 4.1: Agent runtime statistics for 247 agents (65.5% of 377 total), each analyzing 5 extensions.

Metric	Min	Q1	Median	Mean	Q3	Max
Tokens (per agent)	43.6k	66.0k	77.7k	79.1k	90.7k	133.9k
Time (per agent)	2m 33s	4m 54s	5m 54s	5m 54s	6m 48s	10m 10s
Tokens (per ext)	8.7k	13.2k	15.5k	15.8k	18.1k	26.8k
Time (per ext)	31s	59s	1m 11s	1m 11s	1m 22s	2m 2s

The $3.1\times$ variance in tokens (43.6k to 133.9k) and $4.0\times$ variance in execution time (2m 33s to 10m 10s) demonstrate that extension complexity significantly impacts resource requirements. Simple false positives with clear patterns (e.g., hardcoded backend URLs) consume minimal resources at the lower end, while complex true positives requiring multi-file flow tracing and exploit construction demand significantly more at the upper end. The interquartile range (Q1 to Q3) spans 66.0k to 90.7k tokens and 4m 54s to 6m 48s, representing typical resource consumption for 50% of analyses.

For the complete dataset of 1,887 extensions analyzed across 377 agents, the mean token consumption of 79.1k tokens per agent yields an estimated total of approximately 29.8 million tokens. The mean execution time of 5m 54s per agent translates to approximately 37 hours of total computation time. These metrics demonstrate that LLM-based validation at this scale requires substantial computational resources but remains feasible for large-scale security analysis campaigns.

Chapter 5

Results

This chapter presents validation results for the Chrome Web Store dataset, answering our three research questions: static analysis precision under our refined threat model (Section 5.3, RQ1), LLM-based validation effectiveness (Section 5.4, RQ2), and multi-tool agreement analysis (Section 5.5, RQ3). We begin with detection statistics from static analysis (Section 5.1).

5.1 Static Analysis Detection Results

We applied static analysis to the complete Chrome Web Store dataset, categorizing extensions by analysis outcome (Section 5.1.1) and examining detected vulnerabilities (Section 5.1.2).

5.1.1 Dataset Categorization

Table 5.1 presents the distribution of 197,088 analyzed extensions across four outcome categories.

Table 5.1: CoCo analysis outcomes across the complete dataset of 197,088 extensions.

Category	Count	Percentage	Subcategories
Vulnerable	1,887	0.96%	1,623 no timeout, 264 with timeout
Timeout (no vuln)	13,111	6.65%	10,347 CoCo enforced, 2,764 script enforced
Parser Errors	179	0.09%	177 import errors, 2 generation errors
Successful	181,911	92.30%	—
Total	197,088	100%	—

The majority of extensions (92.30%) completed analysis successfully without detecting vulnerabilities. CoCo flagged 1,887 extensions (0.96%) as potentially vulnerable, forming the primary dataset for subsequent LLM-based and manual validation. Table 5.2 shows the breakdown of these 1,887 vulnerable extensions by timeout behavior.

Among the vulnerable extensions, 1,623 (86.0%) completed analysis within the 600-second timeout budget, while 264 (14.0%) exhausted the timeout yet still produced partial vulnerability reports before termination. This shows a considerable increase in flagged extensions, compared to DoubleX, which flagged 224 extensions on the same dataset [1].

In total 13,111 extensions (6.65%) run into a timeout without, indicating complex code structures that exceeded CoCo’s analysis budget without triggering exploitable taint flows or CoCo getting

Table 5.2: Breakdown of 1,887 vulnerable extensions by timeout behavior.

Timeout Status	Count	Percentage
No Timeout	1,623	86.0%
With Timeout	264	14.0%
CoCo-enforced timeout	215	11.4%
Script-stopped timeout	49	2.6%
Total Vulnerable	1,887	100%

stuck in an infinite loop. Errors were reported in 179 of extensions (0.09%), representing a small subset of the total analyzed set. The errors break down as 177 graph construction failures (98.9%) where CoCo could not resolve complex module dependencies, 2 extension generation failures (1.1%) with malformed extension structure, and 0 JavaScript parsing failures, indicating that CoCo did not catch any parser errors (Section 4.4).

5.1.2 Vulnerability Detection Statistics

Across the 1,887 vulnerable extensions, CoCo detected 57 distinct vulnerability sink types. Table 5.3 presents the most frequently detected sink types by number of unique extensions.

Table 5.3: Top 10 most frequently detected vulnerability sinks across 1,887 CoCo-flagged extensions.

Sink Type	Extensions	% of Dataset
chrome.storage.local.set	868	46.0%
chrome.storage.sync.set	310	16.4%
sendResponseExternal	181	9.6%
fetch (resource)	135	7.2%
localStorage.setItem	116	6.1%
window.postMessage	109	5.8%
XMLHttpRequest (url)	99	5.2%
jQuery.html()	64	3.4%
XMLHttpRequest (post)	57	3.0%
chrome.storage.local.clear	53	2.8%

Storage-related sinks dominate CoCo’s detections, with `chrome.storage.local.set` appearing in 868 extensions (46% of vulnerable extensions) and `chrome.storage.sync.set` in 310 extensions. This high prevalence reflects both legitimate extension functionality (storing user preferences, authentication tokens) and genuine security concerns when attacker-controlled data reaches these privileged storage APIs and are then used for cross site scripting or fingerprinting the user [13]. The subsequent manual validation (??) distinguishes between these cases by applying our exploitability criteria.

5.2 CoCo 2021 Baseline Validation

To validate our threat model refinements and establish a baseline, we manually analyzed the 39 extensions from CoCo’s original 2021 paper [16, 15] and compared our results with the LLM classifications, measuring precision under refined exploitability criteria (Section 5.2.1) and LLM validation accuracy (Section 5.2.2).

5.2.1 Manual Analysis Results

Of the 39 extensions in CoCo’s original dataset, only 37 had their results present on GitHub and the developers were informed via email when this was discovered, with the data still not being provided at the time of writing. Manual analysis classified 27 as true positives and 10 as false positives, yielding 73.0% precision, showing that our threat model is different than the one used by the developer.

The dominant false positive pattern involves hardcoded backend infrastructure: attacker message \rightarrow storage.set \rightarrow storage.get \rightarrow fetch(hardcodedURL). While attackers can write data to extension storage that gets sent to developer-hardcoded server, without demonstrated exploitable impact in how the extension processes backend responses, these flows represent trusted infrastructure communication rather than extension vulnerabilities.

5.2.2 LLM Validation on 2021 Dataset

Table 5.4 compares manual and LLM classifications on the 37 valid extensions from the 2021 dataset.

Table 5.4: Manual vs LLM classification agreement on CoCo 2021 dataset (37 extensions).

Classification Agreement	Count	Percentage
Agreement		
Both TP	18	48.6%
Both FP	10	27.0%
Total Agreement	28 / 37	75.7%
Disagreement		
Manual TP, LLM FP (Miss)	9	24.3%
Manual FP, LLM TP (Over-report)	0	0.0%
Total Disagreement	9 / 37	24.3%

The LLM achieved 75.7% agreement with manual analysis (28/37 matching classifications), correctly identifying 18 true positives and 10 false positives. However, the LLM missed 9 genuine vulnerabilities (24.3% false negative rate) while never over-reporting false positives as vulnerable. This same conservative bias appears in the 2025 dataset analysis (??), suggesting it reflects systematic limitations in LLM-based code analysis rather than dataset-specific factors. The LLM’s reported precision on this dataset (48.6%, 18 TP / 37 total) substantially underestimates the true precision of 73.0% (27 TP / 37 total) established by manual analysis. This reveals the need for manual analysis and the limitations of an LLM to correctly identify complex vulnerable extensions.

5.3 CoCo Precision (RQ1)

We performed manual analysis on 239 extensions from the 1,887 CoCo-flagged dataset to measure precision under our refined threat model. This section presents precision measurements (Section 5.3.1), characteristic patterns of genuine vulnerabilities (Section 5.3.2), and dominant false positive patterns (Section 5.3.3).

5.3.1 Manual Validation Results

Manual analysis on a statistically valid sample (239 extensions, 90% confidence, $\pm 5\%$ margin) classified 87 extensions (36.4%) as true positives and 152 (63.6%) as false positives. Table 5.5 presents the classification breakdown.

Table 5.5: Manual classification of 239 CoCo-detected vulnerabilities from 2025 dataset.

Classification	Count	Percentage
True Positive (TP)	87	36.4%
False Positive (FP)	152	63.6%
Total	239	100%

CoCo achieves 36.4% precision (90% CI: 32%–42%) under our refined threat model treating hardcoded backends as trusted infrastructure, requiring complete storage exploitation chains, and focusing on privilege escalation achievable through extension code alone.

5.3.2 Common True Positive Patterns

Manual analysis identified genuine vulnerabilities demonstrating complete exploitation chains. We present two representative examples illustrating common patterns.

Code Execution via Message Passing Listing 5.1 demonstrates attacker-controlled data flowing from `window.postMessage` to `chrome.tabs.executeScript`.

```
1 // Content script - receives attacker message
2 window.addEventListener( message , (event) => {
3   // Forwards to background script
4   chrome.runtime.sendMessage(event.data);
5 });
6
7 // Background script - executes attacker code
8 chrome.runtime.onMessage.addListener((request, sender) => {
9   if (request.sender === 'main' && request.script_requests) {
10     request.script_requests.forEach((script_info) => {
11       chrome.tabs.executeScript(sender.tab.id, script_info); // ← code exec
12     });
13   }
14 });
```

Listing 5.1: Code execution (extension `akhomcaccpndpgckgpkmcijkimphhmk`)

This pattern demonstrates the complete chain: external trigger → message passing → code execution via `executeScript`.

Storage Oracle Listing 5.2 shows an extension allowing attackers to write and read arbitrary storage data via `window.postMessage`.

```
1 // Content script listens on all pages (*://*/*)
2 addEventListener( message , (msg) => {
3   if (msg.data.publicateBrowserExtToken) {
4     // Attacker writes to storage
5     chrome.storage.sync.set({
6       accessToken: msg.data.publicateBrowserExtToken, // ← poisoned
7       token_exp: msg.data.expires
8     });
9   } else if (msg.data.storeGet == accessToken ) {
10    // Attacker reads from storage
11    chrome.storage.sync.get([ accessToken ], (data) => {
12      postMessage({ storeData: data }, * ); // ← exfiltration
13    });
14  }
15 });
```

Listing 5.2: Storage oracle (extension adahoneonjbcdnbkdngadoffhdekhnf)

This pattern enables storage poisoning and information disclosure. Attackers can replace legitimate tokens with malicious ones and retrieve stored authentication data via `postMessage`.

5.3.3 Common False Positive Patterns

Manual analysis revealed recurring patterns where CoCo-detected flows do not constitute exploitable vulnerabilities under our threat model. We present representative examples illustrating common false positive categories.

Attacker Data to Hardcoded Backend Our threat model treats hardcoded developer backend URLs as trusted infrastructure (Section 3.1.3). Listing 5.3 shows attacker-controlled data sent to the developer’s backend server rather than flowing back to the attacker.

```
1 // Background: External message from *.screenjar.com stores JWT
2 chrome.runtime.onMessageExternal.addListener((request, sender, sendResponse) => {
3   chrome.storage.sync.set({ token: request.jwt }); // Attacker JWT stored
4 });
5
6 // Popup: Retrieves JWT and sends to developer's backend
7 chrome.storage.sync.get( token , (result) => {
8   fetch( https://app.screenjar.com/api/check-chrome-login/ , {
9     headers: { Authorization: JWT + result.token } // To hardcoded backend
10  });
11 });
12 // Flow: External message -> storage -> hardcoded backend (not back to attacker)
```

Listing 5.3: Data to trusted backend (extension eannaomdbmcakllbmhhaiaekahjdjhffa)

Hardcoded Backend Data to Storage CoCo flags data from developer backends stored locally. Listing 5.4 shows configuration fetched from the developer’s infrastructure and cached, which does not enable attacker exploitation.

```
1 // Background: Fetch OAuth token from Twitch API
2 const twitchAuthURL = 'https://id.twitch.tv/oauth2/token';
3 fetch(twitchAuthURL, { method: 'POST' })
4   .then(r => r.json())
5   .then(result => {
```

```

6   chrome.storage.sync.set({
7     token: result.access_token, // Store token from trusted API
8     expires: new Date().getTime() + result.expires_in
9   });
10  });
11  // Flow: Hardcoded backend -> storage (no attacker control)

```

Listing 5.4: Backend to storage (extension dophpcobbjngempeadfhkjleblemfjll)

Storage for Internal Boolean Logic Listing 5.5 demonstrates storage poisoning where stored values are only used for boolean checks, not exploitable operations that benefit the attacker.

```

1  // Background: External message from tailyai.co poisoning storage
2  chrome.runtime.onMessageExternal.addListener((request, sender, sendResponse) => {
3    chrome.storage.sync.set(request); // Attacker stores arbitrary data
4  });
5
6  // Content script: Retrieved only for boolean check
7  chrome.storage.sync.get([ status ], (result) => {
8    isLoggedIn = (result.status !== paused ); // Boolean comparison only
9    // No path back to attacker - data not sent via sendResponse or postMessage
10 });
11 // Flow: External message -> storage -> boolean check (incomplete exploitation)

```

Listing 5.5: Storage for boolean logic (extension doglbfdapmngcaplipnjkmebembonbcp)

CoCo Framework Instrumentation Artifacts CoCo reports only framework code line numbers, not actual extension code, when taint flows exist solely in instrumentation. Listing 5.6 shows CoCo detecting `Document_element_href` → `JQ_obj_html_sink` and reporting only Line 20 (framework code). The extension benignly rearranges existing page content for navigation, but CoCo cannot identify which extension code triggered the flow.

```

1  // CoCo detection report (used_time.txt):
2  // from Document_element_href to JQ_obj_html_sink
3  // Line 20: this.href = 'Document_element_href';
4  // No extension code lines reported - only framework line
5
6  // CoCo framework (Line 20): Models DOM elements with taint source
7  function Document_element(id, class_name, tag) {
8    this.href = 'Document_element_href'; // Reported line
9    MarkSource(this.href, 'Document_element_href');
10 }
11
12 // Extension code (Line 467+): Actual code unreported by CoCo
13 $(document).ready(function() {
14   var firstChild = $( '#wikicontent' ).children( :first );
15   if (firstChild[0].nodeName == UL ) { // DOM element access
16     $( '#wikicontentFly' ).html( $( '#wikicontent ul:eq(0)' ));
17   }
18 });

```

Listing 5.6: CoCo reports framework line only (extension fciaioepfedpcpkpggjmanibleblje)

5.4 LLM-Based Validation (RQ2)

We performed LLM-based validation on all 1,887 CoCo-flagged extensions using Claude Sonnet 4.5 agents (Section 4.5). This section presents classification results (Section 5.4.1), agreement with

manual ground truth (Section 5.4.2), and systematic classification errors (Section 5.4.3).

5.4.1 Classification Results

Table 5.6 presents the aggregate classification results from the LLM-based validation workflow.

Table 5.6: LLM-based classification of CoCo-detected vulnerabilities across the complete 2025 dataset.

Classification	Count	Percentage
True Positive (TP)	461	24.4%
False Positive (FP)	1,426	75.6%
Total Analyzed	1,887	100%

The LLM classified 461 extensions (24.4%) as true positives and 1,426 (75.6%) as false positives. However, comparison against manual ground truth reveals systematic conservative bias: the LLM’s 24.4% precision substantially underestimates the actual 36.4% precision measured through manual analysis (Section 5.3.1). This 12 percentage point gap demonstrates that LLM-based validation, while scalable, systematically under-reports genuine vulnerabilities.

5.4.2 Agreement with Manual Ground Truth

Comparing LLM classifications against manual analysis on 239 extensions reveals 81.6% agreement (195 matching classifications). Table 5.7 presents the agreement breakdown.

Table 5.7: Agreement between manual analysis and LLM classifications on 239 extensions.

Classification Agreement	Count	Percentage
Agreement		
Both TP	48	20.1%
Both FP	146	61.1%
Total Agreement	195 / 239	81.6%
Disagreement		
Manual TP, LLM FP (Miss)	38	15.9%
Manual FP, LLM TP (Over-report)	6	2.5%
Total Disagreement	44 / 239	18.4%

The disagreement distribution reveals pronounced asymmetry: the LLM missed 38 genuine vulnerabilities (43.7% FNR) while over-reporting only 6 false positives (3.9%). This demonstrates extreme conservative bias. The LLM exhibits high specificity (96.1%—correctly identifying 146 of 152 benign flows) but lower sensitivity (55.2%—correctly identifying only 48 of 87 genuine vulnerabilities).

5.4.3 LLM Classification Errors

Analysis of the 38 missed vulnerabilities reveals four systematic failure modes (Table 5.8), distinct from CoCo’s false positive patterns (Section 5.3.3) which describe characteristics of benign flows.

Failure Mode	Count	%
Code Search Failures	12	31.6%
Threat Model Reasoning Errors	10	26.3%
Code Complexity Barriers	9	23.7%
CoCo Report Limitations	7	18.4%
Total	38	100%

Table 5.8: LLM failure in missing genuine vulnerabilities (38 cases).

Code Search Failures (31.6%) The LLM failed to locate key vulnerability chain components across files. In 9 of 12 cases, the LLM identified attacker-controlled data stored via `chrome.storage.set` but missed corresponding `get` operations retrieving this data, representing fundamental cross-file dependency tracking limitations.

Threat Model Reasoning Errors (26.3%) The LLM identified complete data flows but misunderstood exploitability. Examples include treating storage oracle attacks as safe (“attacker retrieves own data”) or inappropriately trusting `externally_connectable` domain restrictions.

Code Complexity Barriers (23.7%) Minified code (5 cases) or complex data flow patterns (4 cases) prevented the LLM from tracing flows.

CoCo Report Limitations (18.4%) Seven cases reflect CoCo reporting issues rather than LLM failures—the LLM restricted analysis to CoCo-reported instrumented lines or encountered edge cases.

These failures demonstrate that while LLMs effectively identify common vulnerability patterns, they face fundamental challenges in multi-file dependency tracking, complex data flow analysis, and nuanced threat model reasoning. The 43.7% false negative rate indicates LLM-based analysis serves best as a triage mechanism rather than standalone validation. Detailed analysis with code examples is presented in [Section 5.4.2](#).

5.5 Multi-Tool Agreement (RQ3)

We compared CoCo against DoubleX on the same 2025 Chrome Web Store dataset, to assess whether multi-tool agreement indicates higher-confidence detections. This section presents overlap analysis ([Section 5.5.1](#)) and precision improvements ([Section 5.5.2](#)).

5.5.1 Overlap Analysis

Adam [1] applied DoubleX to the 2025 dataset, flagging 224 extensions with 56.7% precision (127 TP, 97 FP). The overlap with CoCo comprises 74 extensions—33.0% of DoubleX detections but only 3.9% of CoCo’s 1,887 detections ([Figure 5.1](#)).

This asymmetry reflects different detection philosophies: CoCo’s comprehensive source/sink modeling produces broader coverage while DoubleX’s stricter exploitability criteria yield more selective detections.

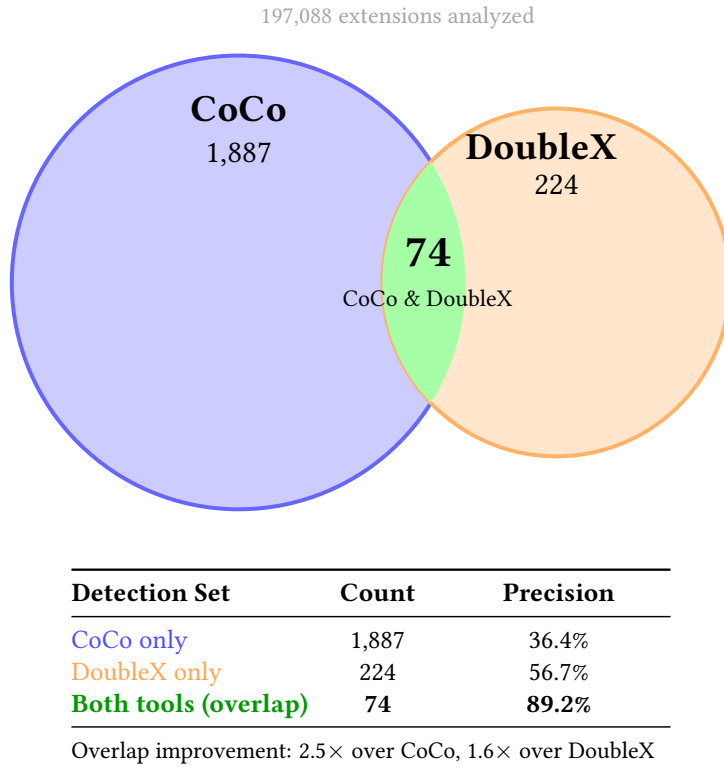


Figure 5.1: CoCo and DoubleX detection overlap.

5.5.2 Precision Improvement

Manual analysis of the 74 extensions revealed 89.2% precision (66 TP, 8 FP)—a $2.5\times$ improvement over CoCo’s 36.4% and $1.6\times$ over DoubleX’s 56.7%. Multi-tool agreement filters tool-specific quirks and highlights patterns that multiple independent analyses identify as concerning. LLM validation on the 74-extension overlap demonstrated substantially improved performance relative to its results on the complete CoCo dataset. [Table 5.9](#) compares LLM performance metrics between the overlap and overall dataset. The improved performance indicates that multi-tool detections contain more straightforward exploitation patterns aligned with LLM reasoning capabilities, where clearer vulnerability chains reduce the code search failures and complexity barriers that plagued LLM analysis of the broader dataset.

Metric	Overall (239 ext.)	Overlap (74 ext.)
Agreement with Manual	81.6% (195/239)	87.8% (65/74)
Sensitivity (Recall)	55.2% (48/87)	86.4% (57/66)
Specificity	96.1% (146/152)	100% (8/8)
False Negative Rate	43.7% (38/87)	13.6% (9/66)
False Positive Over-reports	3.9% (6/152)	0% (0/8)

Table 5.9: LLM performance comparison: overlap vs. overall dataset.

Chapter 6

Conclusion

In this thesis, we systematically validated CoCo’s vulnerability detections on the contemporary Chrome Web Store, investigating static analysis precision and automated validation at scale. We established a threat model requiring flows to demonstrate direct exploitability within extension code, treating hardcoded developer infrastructure as trusted and requiring complete exploitation chains for storage operations. Manual analysis of 239 systematically sampled extensions revealed 36.4% precision (90% CI: 32–42%), with three dominant false positive patterns accounting for 74.3% of misclassifications: hardcoded backend communications, incomplete storage chains, and attacker parameters to benign APIs.

We evaluated whether LLM-based agents can automate vulnerability validation by implementing a parallel sub-agent analysis system using Claude Sonnet 4.5. The LLM achieved 81.6% agreement with manual ground truth, revealing complementary strengths and limitations. LLMs excel at identifying false positives flagged by static analysis tools (96.1% specificity), effectively filtering obvious benign patterns like hardcoded backend communications and incomplete storage chains. However, LLMs miss hidden vulnerable patterns, achieving only 55.2% sensitivity due to cross-file tracking failures (31.6%), threat model reasoning errors (26.3%), and code complexity barriers (23.7%). The extreme conservative bias—38 false negatives versus 6 false positive over-reports—demonstrates that manual analysis remains essential not only for validation quality assurance but also for discovering hidden vulnerability patterns that can then be encoded into the LLM methodology to improve future automated classifications.

Multi-tool agreement between CoCo and DoubleX yielded 89.2% precision on 74 jointly-flagged extensions, representing a 2.5× improvement over CoCo’s baseline 36.4% and a 1.6× improvement over DoubleX’s 56.7%. This demonstrates that complementary static analysis approaches filter tool-specific false positives and identify clearer exploitation patterns. LLM validation performance also improved on this subset (87.8% agreement, 86.4% sensitivity), suggesting automated triage can reliably process high-confidence detections.

Our findings establish empirical foundations for deploying static analysis at scale with LLM-assisted validation. LLMs serve as effective first-pass filters for false positives, but the iterative workflow requires manual analysis to (i) validate LLM classifications, (ii) discover hidden vulnerability patterns missed by automated analysis, and (iii) refine the LLM methodology with newly-identified patterns. This human-in-the-loop approach leverages LLM strengths in pattern recognition while relying on manual expertise to continuously improve the automated system’s understanding of subtle exploitation scenarios. The adaptable nature of LLM-based validation offers significant advantages as exploitation techniques evolve—newly-discovered attack patterns can be rapidly integrated into the methodology prompt without modifying static analysis tools,

enabling the validation system to keep pace with the evolving threat landscape.

Future work could pursue tighter integration between static analysis output and LLM validation. Clearer output marking complete data flow paths, from source through intermediate steps to sinks, would provide more structured input for flow reconstruction. A stricter threat model applied during static analysis, filtering hardcoded backend flows and incomplete storage chains before LLM validation, would reduce ambiguous cases requiring interpretation. An improved methodology prompt incorporating patterns from manual analysis of LLM misclassifications might further increase reliability and address the systematic reasoning errors we observed.

Bibliography

- [1] Patrick Jan Adam. Are browser extensions still vulnerable? revisiting doublex on the latest chrome web store data. Bachelor’s thesis, Ludwig-Maximilians-Universität München, July 2025. Advisor: Matías Gobbi.
- [2] Mohammad M. Ahmadpanah, Matías F. Gobbi, Daniel Hedin, Johannes Kinder, and Andrei Sabelfeld. CodeX: Contextual flow tracking for browser extensions. In *Proc. ACM Conf. Data and Application Security and Privacy (CODASPY)*, pages 1–15. ACM, 2025. doi: 10.1145/3714393.3726495.
- [3] Anthropic. Introducing claude sonnet 4.5, September 2025. URL <https://www.anthropic.com/news/claude-sonnet-4-5>. Accessed: 2025-01-10.
- [4] Aurore Fass, Doliere Francis Some, Michael Backes, and Ben Stock. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS ’21)*, pages 1789–1804, Virtual Event, Republic of Korea, 2021. ACM. doi: 10.1145/3460120.3484745.
- [5] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. Security weaknesses of copilot-generated code in github projects: An empirical study. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–37, 2024. doi: 10.1145/3716848.
- [6] Google. Chrome web store statistics, 2024. URL <https://chrome-stats.com/chrome/stats>. Accessed: 2025-01-10.
- [7] Google Chrome Developers. externally_connectable — chrome extensions, 2025. URL <https://developer.chrome.com/docs/extensions/reference/manifest/externally-connectable>. Accessed: 2025-01-21.
- [8] Google Chrome Developers. Manifest v2 support timeline — chrome extensions, 2025. URL <https://developer.chrome.com/docs/extensions/develop/migrate/mv2-deprecation-timeline>. Accessed: 2025-01-10.
- [9] Google Chrome Developers. Welcome to manifest v3 — chrome extensions, 2025. URL <https://developer.chrome.com/docs/extensions/develop/migrate/what-is-mv3>. Accessed: 2025-01-10.
- [10] Google Chrome Developers. Remotely hosted code — chrome extensions, 2025. URL <https://developer.chrome.com/docs/extensions/develop/migrate/improve-security#remove-remote-code>. Accessed: 2025-01-10.
- [11] Google Chrome Developers. Service workers in extensions — chrome extensions, 2025. URL <https://developer.chrome.com/docs/extensions/develop/concepts/service-workers>. Accessed: 2025-01-10.
- [12] Python Software Foundation. signal — set handlers for asynchronous events, 2025. URL <https://docs.python.org/3/library/signal.html>. Accessed: 2025-01-21.

- [13] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. Extension breakdown: Security analysis of browsers extension resources control policies. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security '17)*, pages 679–694, Vancouver, BC, Canada, 2017. USENIX Association. URL <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/sanchez-rola>.
- [14] Stack Overflow. 2024 stack overflow developer survey: Ai, 2024. URL <https://survey.stackoverflow.co/2024/ai>. Accessed: 2025-07-22.
- [15] Jianjia Yu, Song Li, Junmin Zhu, and Yinzhi Cao. Coco: Coverage-guided, concurrent abstract interpretation for browser extension vulnerability detection. <https://github.com/Suuuuuzy/CoCo>, 2023. Accessed: 2025-08-12.
- [16] Jianjia Yu, Song Li, Junmin Zhu, and Yinzhi Cao. Coco: Efficient browser extension vulnerability detection via coverage-guided, concurrent abstract interpretation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS '23*, pages 2441–2455, New York, NY, USA, 2023. ACM. doi: 10.1145/3576915.3616584.