CoCo: Efficient Browser Extension Vulnerability Detection via Coverage-guided, Concurrent Abstract Interpretation

Jianjia Yu
Johns Hopkins University
Baltimore, MD, USA
jyu122@jhu.edu

Song Li†
Zhejiang University
Hangzhou, Zhejiang, China
songl@zju.edu.cn

Junmin Zhu†
Shanghai Jiao Tong University
Shanghai, China
junmin.zhu@sjtu.edu.cn

Yinzhi Cao
Johns Hopkins University
Baltimore, MD, USA
yinzhi.cao@jhu.edu

ABSTRACT

Extensions complement web browsers with additional functionalities and also bring new vulnerability venues, allowing privilege escalations from adversarial web pages to use extension APIs. Prior works on extension vulnerability detection adopt classic static analysis, which is unable to handle dynamic JavaScript features such as those function calls as part of array lookups. At the same time, prior abstract interpretation focuses on lightweight server-side JavaScript, which often cannot scale to client-side extension code due to object explosions in the abstract domain.

In this paper, we design, implement and evaluate a novel, coverage-driven, concurrent abstract interpretation framework, called CoCo, to efficiently detect vulnerabilities in browser extensions. On one hand, CoCo parallelizes abstract interpretation with concurrent taint propagation for each branching statement, message passing and content/background scripts to detect vulnerabilities with improved scalability. On the other hand, CoCo prioritizes analysis that increases code coverage, thus further detecting more vulnerabilities. Our evaluation shows that CoCo detects at least 43 zero-day, exploitable, manually-verified extension vulnerabilities that cannot be detected by state-of-the-art works. We responsibly disclosed all the zero-day vulnerabilities to extension developers.

CCS CONCEPTS

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

KEYWORDS

JavaScript, Browser Extension, Security

ACM Reference Format:
Jianjia Yu, Song Li†, Junmin Zhu†, and Yinzhi Cao. 2023. 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), November 26–30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616584

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3616584

1 INTRODUCTION

Browser extensions are personalized add-ons written in JavaScript to native browsers to boost native browsers' functionalities. Popular browser extensions are often installed by millions of users for daily usage. For example, Grammarly [6] provides grammatical suggestions for its users, and Google scholar [5] provides quick searches and citations of academic papers. Because browser extensions need to provide additional functionality close to the native browser, they often have a higher privilege than the normal website visited by web users. For example, browser extensions may send cross-origin requests beyond the restriction enforced by the same-origin policy and access browser-only storage such as bookmarks and browsing history. Such privileged APIs need to be protected from normal web pages in preventing privilege escalations.

To enforce such protection, modern browser extension architecture, particularly Google Chrome, adopts a so-called isolated world [1] to separate scripts (called background or more recently replaced by service workers in manifest V3) having access to privileged APIs from those (called content scripts) that are close to the potential adversary, i.e., the web page. Scripts from these two worlds communicate with each other via a message-passing channel. While such an architecture greatly reduces potential vulnerabilities, still the communication between two types of scripts will lead to the access of privileged APIs and data as shown in many real-world vulnerabilities [4, 11] in the wild.

Researchers have been working on the detection of privilege escalations. On one hand, prior works [14, 35, 55] that detect browser extension vulnerabilities propose using static analysis that tracks dataflow between adversary-controlled inputs (e.g., a message sent to the background script) and a privileged API. Specifically, EmPoWeb [55]—the first work of its kind in modern extension vulnerability detection—combines call graph analysis and manual inspection to find hundreds of vulnerable extensions. A follow-up work called DoubleX [35], improves the static analysis performed by EmPoWeb with an Extension Dependence Graph (EDG) for more

† The two authors contribute to the paper when they are either studying or interning at Johns Hopkins University.

accurate yet automated detection. However, neither work is able to handle dynamic JavaScript features, such as function calls that are resolved dynamically (e.g., `funcs[name]()` where `name` is another variable).

On the other hand, recent advances in server-side vulnerability detection [39, 42, 43, 46] use abstract interpretation to resolve aforementioned dynamic JavaScript features. For example, ODGen [43] interprets Node.js packages in an abstract domain, called Object Dependence Graph, and queries the graph for dynamic object resolution. However, existing server-side JavaScript abstract interpretation is single threaded, which cannot detect vulnerabilities related to concurrency features of client-side extensions, such as isolated worlds and message passing. Furthermore, while abstract interpretation itself is promising, it often cannot scale especially given the size and complexity of client-side browser extension code. One common issue is that the number of objects explodes in the abstract domain, which prevents abstract interpretation from even reaching vulnerable code, thus leading to many false negatives.

In this paper, we design a novel, coverage-guided, concurrent abstract interpretation framework, called CoCo, on a graph-based abstract domain to efficiently detect browser extension vulnerabilities using static analysis. The key insights of CoCo are two-fold as suggested by the two "Co"s in its name. On one hand, CoCo parallelizes abstract interpretation with concurrent taint propagation by converting each branching statement, asynchronous callback functions, and content/background scripts with message passing into multiple threads to better detect vulnerability with improved scalability. For example, the concurrent taint analysis propagates taint information among different threads across message passing, to detect browser extension vulnerabilities. Specifically, CoCo models and maintains event loops to iterate through callback functions for content and background scripts while keeping track of taints. Then, CoCo also simulates message passing between content and background scripts and propagates taint information across scripts during concurrent abstract interpretation.

On the other hand, CoCo prioritizes abstract interpretation for unseen code to maximize code coverage, thus further increasing the number of detected vulnerabilities. Specifically, CoCo allocates analysis time to each thread based on the branching level and past performance in increasing code coverage. That is, CoCo allocates more time to a thread if it either analyzes more code or is located on a top-level branch. Then, CoCo preempts a thread after it uses up its allocated time. Furthermore, because the number of threads and abstract domain states could be exponential given many branching statements in a target extension, CoCo merges the abstract domain, particularly the graph representation, of these different threads after the conditional statement to reduce the number of states and threads and avoid state explosion. That is, CoCo reduces two threads into one by keeping newly-added nodes or edges if they exist in either thread and removing nodes or edges if they are deleted from both threads.

We implemented a prototype of CoCo, which is available at this anonymous repository [9]. Then, we crawled 145K+ extensions from the Chrome extension store for evaluation. Our evaluation shows that CoCo finds at least 43 zero-day, exploitable vulnerabilities that cannot be detected by prior works (e.g., DoubleX [35] and a modified version ODGen [43], called ODGen-ext, to detect extension vulnerabilities) and verified manually. We responsibly disclosed all our findings to extension developers and so far have received one confirmation. We compared CoCo with DoubleX, the state-of-the-art extension vulnerability detection tool as well as ODGen-ext. Our evaluation shows that CoCo outperforms both DoubleX and ODGen-ext in terms of the number of detected vulnerabilities and false positives/negatives.

2 BACKGROUND

In this section, we present some background knowledge on browser extension architecture and abstract interpretation.

Browser extension architecture. Modern browser extension often adopts an isolation mechanism, e.g., isolated world [1] in Chrome, to prevent scripts with a low privilege to access higher-privileged APIs. For example, such isolation divides scripts in Google Chrome (including extensions and web pages) into three types: web page scripts, content scripts, and background (service workers in manifest V3) scripts.

Since different scripts are isolated from each other, a communication mechanism is necessary to enable the exchange of information. Specifically, such communication is facilitated by message passing in browser architecture and we list the following four types of message passing mechanisms.

- Web page ↔ Content script. Web page script communicates with content scripts via a regular `postMessage` channel with `addEventListener` and `onmessage` APIs.
- Content ↔ Background script. There are two types of communications between content and background scripts (or service worker in V3). First, they can communicate via one-time requests APIs, i.e., `sendMessage` and `onMessage` under either `runtime` (content) or `tabs` (background). Such communication exchanges one message at a time. Second, they can communicate via long-lived APIs (e.g., `connect` and `onConnect`) to exchange multiple messages.
- Web page ↔ Background script. A web page can communicate with a background script if permissions are declared, i.e., the `externally_connectable` field of the manifest file contains the web page's URL. The communication is similar to content ↔ background script in the two types with the exception that `onMessageExternal` and `onConnectExternal` are used.
- Extension ↔ Another extension. Such a communication is similar to web page ↔ background script and enabled by default. A whitelist can be declared in the manifest file using allowed extensions' IDs.

Abstract interpretation. Abstract interpretation is a technique to approximate the execution of a given computer program upon an abstract domain without concrete inputs. There are two types of abstract interpretation in the literature based on the abstract domain types, which are lattice- and graph-based. First, classic abstract interpretation [29] adopts a lattice structure as the abstract domain. One challenge is the over-approximation of abstract values and thus many prior works propose optimizations, such as trace partitioning [44, 53], to improve traditional interval analysis by moving some statements outside a branching statement inside.

Second, recent works [42, 43] propose graph-based abstract interpretation for vulnerability detection given efficient graph operations on the abstract domain. One challenge is scalability given the exponential number of objects in the graph. CoCo works on graph-based abstract interpretation to improve their scalability, which is different from existing optimizations. That is, existing optimization methods—which operate on a lattice-based abstract domain—are not applicable to graph-based abstract interpretation. Take trace partitioning for example, which merges intervals of an abstract value. Instead, CoCo merges graphs from different threads, which is a completely different concept. Similarly, it also remains unknown to apply other static analysis optimizations, like loop unrolling and object packing, for graph-based abstract domains.

3 OVERVIEW

In this section, we present an overview of CoCo with a motivating example, a solution overview and the threat model.

3.1 A Motivating Example

We illustrate a motivating example, called *AliExpress to Shopify Importer*, to describe the challenges of browser extension vulnerability detection. This extension is designed to help users automatically import AliExpress products' information into another website, called Shopify. It has a privilege escalation vulnerability, which allows a website to escalate its privilege to the browser extension, thus sending arbitrary third-party requests regardless of the cross-origin header.

Listing 1 shows the vulnerable code: Line 28 in the `getData` function is where the vulnerability locates. The AJAX call in browser extension is privileged, but the destination `url` at Line 29 is controllable by an adversary from a website via a message. Lines 1–6 show the exploit code: A webpage adversary sends a message with a cross-origin `product_url` and obtains the responses. The challenges of detecting this vulnerability are manifested in two-fold:

- Dynamic Function Call. The invocation of `getData` is via a dynamic object lookup at Lines 17–18. Specifically, say the adversary provides a string `"getData"` in `req.action` like the exploit code does at Line 3. Then, the vulnerable code at Line 17 finds the index of `"getData"` in the `actionList` array at Line 9 to avoid unauthorized function calls (i.e., `actionList[index]` is `"getData"`). Next, Line 18 looks up the function dynamically via `window["getData"]` and invokes it with `req` as the parameter. State-of-the-art extension vulnerability detection, namely DoubleX [35], cannot find the dynamic call edge between Line 17 and Line 24, thus skipping the data-flow edge between `req.product_url` at Line 29 and the value at Line 3. The vulnerable extension is included in the EmPoWeb dataset [55] but detected mostly with manual work by the author: The dynamic call edges are also missing in their analysis, but a human expert can identify them.
- Reachability. The invocation of `getData` is located in an `else` branch of an `if` statement at Line 13, leading to a reachability issue of static abstract interpretation. Specifically, state-of-the-art abstract interpretation, such as ODGen [43] and ObjLupAnsys [42], stuck in the `if` branch (Lines 13–15) for this specific extension because of state explosion. Therefore, they cannot even reach Line 17 to resolve the aforementioned dynamic call edge.

```
1  // exploit code located in a web page
2  var editorExtensionId="cfbodcmobhpfbjhbennacnanbmpbcfkd"
3  chrome.runtime.sendMessage(editorExtensionId, {action: "getData"
       , product_url:"https://appfreaker.com/"},
4    function(response) {
5      console.log(response)
6    });
7
8  // vulnerable extension code
9  var actionList = ["addProduct", "getData"];
10 chrome.runtime.onMessageExternal.addListener(
11  function(req, caller, res) {
12   if (req.hasOwnProperty("action")) {
13     if (actionList.indexOf(req.action) == -1) {
14       //dealing with invalid actions
15     } else {
16       //call the callback with request and caller data
17       var index = actionList.indexOf(req.action);
18       res({"result": window[actionList[index]](req)});
19     }
20   } else {
21     //dealing with no actions
22   }
23 })
24 function getData(req) {
25   var result = {};
26   var product_url = req.product_url;
27   result["product_url"] = req.product_url;;
28   $.ajax({ // privileged escalation for sending AJAX request
       bypassing SOP
29     url: req.product_url,
30     type: "get",
31     async: false,
32     success: function(resdata) {
33       result["data"] = resdata;
34       result["status"] = 'success';
35     },
36     error: function(resdata) {
37       result["data"] = resdata;
38       result["status"] = "error";
39     }
40   });
41   result["message"] = "The application is running";
42   return result;
43 }
```

Listing 1: A Motivating example: AliExpress to Shopify Importer (Sink function is at Line 28; an adversary can send privileged AJAX requests bypassing same-origin policy)

3.2 Solution Overview

We describe an overview of CoCo in detecting the vulnerability of our motivating example in Listing 1. From a high-level perspective, CoCo finds a data flow from a user input (i.e., the `req` object at Line 11) to a sensitive function (i.e., the `url` parameter of the `$.ajax` function at Line 28) and finally to the user again (i.e., the parameter of the `res` at Line 18, which is provided by the user at Line 11).

Now let us describe how CoCo solves the aforementioned challenges. First, CoCo resolves the dynamic call edge at Line 18 via looking up objects in the abstract domain. That is, CoCo first resolves `index` as 1, then fetches `actionList[index]` as `"getData"`, and finally looks up the `getData` function via `window["getData"]`. All the information is stored in the abstract domain as nodes and edges.

Second, CoCo solves the reachability issue by scheduling abstract interpretations of different branches in parallel and allocating analysis time by priority values associated with code coverage. Let us use Listing 1 for the explanation. CoCo analyzes two branches of the `if` statement at Line 13 in two threads in parallel. That is, CoCo switches between the analysis of Line 14 and Lines 17–18

Table 1: A list of sensitive sink APIs modelled by CoCo. (Note that AJAX requests in content scripts are not considered sensitive, because their content scripts are subject to the same origin policy according to a new change [2].)

Consequences	Detailed APIs
Code execution	eval, tabs.executeScript, setTimeout, setInterval
AJAX requests	ajax (not content scripts), fetch, get, post, XMLHttpRequest().open
File downloads	downloads.download
Storage access	
- Cookie	cookies.get, cookies.getAll, cookies.set
- Bookmark/history	bookmarks.getTree, bookmarks.create, history.search, history.getVisits
- Other storage	topSites.get, management.getAll, management.setEnabled, storage.local.get, storage.sync.get, storage.local.set, storage.sync.set, storage.local.clear, storage.sync.clear, localStorage.clear, localStorage.setItem

based on an analysis interval. Say CoCo is stuck in a `for` loop or a recursive call in the `if` branch at Line 14. Because the code coverage stays mostly the same, CoCo boosts the priority of the `else` branch (Lines 17–18), which resolves the dynamic call edge between Line 18 and Line 24.

3.3 Threat Model

Our threat model assumes two parties: an adversary and a victim. The adversary is a webpage or another browser extension, which tries to exploit the victim browser extension's vulnerability via two methods. First, the adversary may send messages to the victim extension via message-passing APIs documented in Section 2. Second, the adversary may trigger DOM events that are listened to by the victim extension. For example, the adversary dispatches a new customer event registered by and listened to by the content script of the victim extension by a callback to trigger the vulnerability. The victim is a browser extension that may have a vulnerability. From the high level, our in-scope vulnerabilities can be summarized as *privilege escalation*, i.e., the adversary manipulating the invocation of a privileged API in the victim extension to gain permissions that they do not originally have. A list of such APIs is shown in Table 1, which follows prior works [35, 55] with addition of `bookmarks.create`, `localStorage` and a few removals, i.e., cross-origin requests in content scripts (which are disabled since Chrome 85 [2]). More specifically, we describe those APIs in Table 1 below based on their consequences.

- Code execution: Such APIs allow an adversary to execute code under the privilege of the extension. Therefore, the adversary may access other sensitive APIs.
- Privileged AJAX requests: Such APIs allow an adversary to send cross-origin AJAX requests, essentially bypassing websites' same origin policy (SOP). Note that due to new restrictions, such AJAX request APIs are not considered for content scripts.
- File downloads: Such APIs allow an adversary to download arbitrary files to the user's device, which may further hamper users' operating systems.

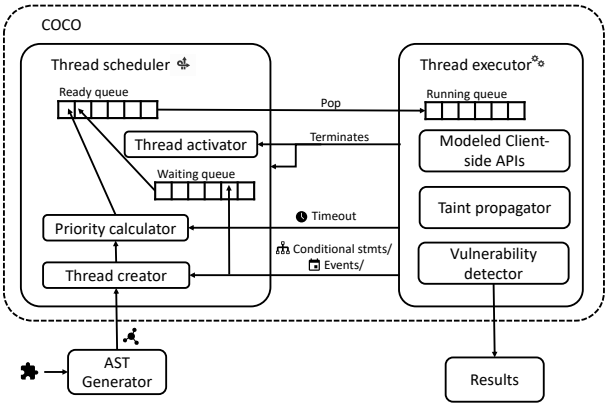

Figure 1: Overall System Architecture of CoCo, which has two major components: thread scheduler (coverage-driven) and thread executor (concurrent abstract interpretation)

- Storage access: Such APIs grant either read or write access to browser or extension storage, such as cookies, history, bookmarks, and extension's local storage. Accessing cookies may lead to session hijacking, session fixation, and user data tampering. Accessing history and bookmarks may be used to track the user's interests and habits. Tampering with `bookmarks.create` may lead to the creation of a seemingly-benign bookmark with a malicious URL.

4 DESIGN

In this section, we start by describing the system architecture in Section 4.1, and then present two components of CoCo in Section 4.3 and 4.2 respectively.

4.1 System Architecture

CoCo accepts the Abstract Syntax Tree (AST) of a given browser extension and outputs whether the extension contains certain vulnerabilities. Figure 1 shows the overall system architecture with two main components: thread scheduler and thread executor. The thread scheduler is responsible for creating threads and scheduling the thread execution based on a priority calculated based on factors such as code coverage and depth. Then, the thread executor analyzes a code snippet using abstract interpretation and interacts with modeled client-side APIs. The analysis mainly contains two parts: taint propagation and vulnerability detection. That is, CoCo propagates taint information from adversary-controlled sources and detects whether it can flows to a sink without sanitization.

Note that the thread scheduler has full control over the thread executor. For example, during execution, a thread can be preempted by the scheduler and replaced with another thread. Then, when a thread finishes execution, the scheduler merges the thread with its parent thread for continued execution. That is, the thread executor is responsible for the analysis and the scheduler is responsible for prioritizing the analysis.

4.2 Thread Executor

The thread executor of CoCo abstractly interpret a given code snippet and generates states (particularly, object dependence graph [43])

Table 2: Annotations of procedures, sets, constants, and contexts used in the operational semantics in Figure 2

Name	Description
Context	Context variables
Δ	Mapping an object to its taint set
Σ	Mapping an event to its callback
Θ	Mapping a message port to its callback on receiving a message
Procedures(P)	Graph operations
$\Phi(x)$	Locating the object for a variable x
$\Lambda(f)$	Locating the corresponding event of a function f
Copy(x)	Copying the x variable
New(e)	Creating a new variable for the expression e
Port(x)	Creating a long-lived connection port with variable x
LkupPort(f)	Finding the corresponding port for function f
Sets(S)	*Sets defined and used by CoCo*
F	Sanitization functions
Set Operation	*Set operation defined and used by CoCo*
$\oplus$	$\Delta[x \oplus f]$ is to append a sanitation function f to each of the list $[ts_k, f_{k_1}, ..., f_{k_n}]$ in set $\Delta[x]$, return the updated mapping Δ
Variables(V)	*Constants/built-in functions/variables of extensions*
S	Sender variable, constant
R	Built-in response function for simple one-time requests
*	Wildcard variable

in an abstract domain. CoCo performs three main tasks in the executor: (i) taint propagation, (ii) vulnerability detection, and (iii) interaction with modeled client-side APIs.

4.2.1 Taint Propagation. CoCo defines a taint as a list with the first item as the taint source followed by a list of sanitization functions between the source and a current object (if there is any). Specifically, x is an object, and $\Delta[x]$ maps the object x to its taint set. Each taint in the taint set follows the format like $[ts_k, f_{k_1}, ..., f_{k_n}]$, where ts_k is the kth taint source, $f_{k_1}, ..., f_{k_n}$ are the sanitization functions. The taint is stored together with all the abstract objects in the abstract domain. Note that we use ts_k because there might exist several taints for one given object.

First, we present different annotations of the operational semantics of the taint analysis in Table 2. A state σ in CoCo's taint analysis is represented by a tuple (Δ, Σ, Θ), where Δ is a mapping between objects and taint set, Σ a mapping between events and callbacks, and Θ a mapping between message ports and callbacks. Furthermore, we denote the update of an object x with a taint t as $x \leftarrow t$. That is, $\Delta[x \leftarrow t]$ means that CoCo taints the object a with t and updates the states. Similarly, we denote the appendment of a sanitization function f to a taint $\Delta[x]$ as $\Delta[x \oplus f]$. For the reason of space, all other annotations are listed in Table 2.

Next, we describe the taint propagation process. Figure 2 shows the operational semantics, which can be generally classified into three categories: basic expressions, simple one-time requests, and long-lived connections. The latter two are two types of message-passing mechanisms. The inference rules in the operational semantics define the valid transitions of a composite piece of syntax in terms of the transitions of its components [10]. Each inference rule can be seen as an *if-then* pair where the upper part corresponds to the *if* condition and the lower part corresponds to the *then* action.

First, let us describe the "basic expressions" in Figure 2 below:

- Taint init. This rule initializes taint variables.
  Condition: The initial state of CoCo is σ, which can be represented as a tuple (Δ, Σ, Θ), the taint of source variable src is t ($\Phi(src)$ corresponds to the t_{s_k} in $[ts_k, f_{k_1}, ..., f_{k_n}]$).
  Action: The expression get_taint(src) in state σ is reduced to a new state, i.e., $(\Delta[\Phi(src) \leftarrow t], \Sigma, \Theta)$, where $\Delta[\Phi(src) \leftarrow t]$ denotes updating the taint of variable src to t.
- Property access. This rule propagates taints for property access.
  Condition: The state σ is represented as a tuple (Δ, Σ, Θ). A property of a tainted object is accessed through $x[p]/x.const$. CoCo tries to resolve the property and represents the property object as p or p_{new} if the property can or can not be resolved, respectively.
  Action: There are two cases to resolve the property access, which are $x[p]$ and $x.const$. If the property can be resolved, CoCo does not propagate the taint. Otherwise, if a property of a tainted object is accessed and *cannot* be resolved, CoCo propagates the taint from the object to the property object. The reason is that if a property cannot be resolved, it is likely coming from an adversary, e.g., a JSON object with an inner structure.
- Binary operators. This rule deals with the binary operators.
  Condition: The two operands under state σ can be reduced to two different new states. The new variable generated by the operation is denoted as x_{new}. CoCo updates the taint of x_{new} with the union of the taint sets of its two operands, namely, $\Delta_1[\Phi(x_1)]$ and $\Delta_2[\Phi(x_2)]$.
  Action: The binary operator under state s is reduced to a new state with the new taint, the union of event mapping and the union of the message port mapping.
- Function calls. This rule deals with the function calls.
  Condition: The state σ is represented as the tuple as before. The returned new variable (if any) is denoted as x_{ret}. A possible new mapping is denoted as Δ'. The update of Δ' is to update the taint of x_{ret} with the union of the taints of all the parameters appended with the called function f.
  Action: There are three cases. (i) If the function definition can be resolved, CoCo goes into the function body so that the function call is reduced to the function body statements, e.g., binary operators and property access. No update of taint is needed by the function call syntax. (ii) If the function is a sanitization call and cannot be resolved, the taint mapping Δ is updated to Δ', propagating the taint from the parameters to the returned object and appending the sanitization function. (iii) If the function definition is not a sanitization function and can not be resolved, CoCo propagates the taint. The taint mapping updates to Δ''.

Second, we describe taint propagation in "simple one-time requests" in Figure 2, which sends a one-time JSON-serializable message between the content script and the background script (or service worker in V3). When analyzing `sendMessage`, CoCo copies the message and propagates taints from the sender to the receiver. If an optional callback [8] is enabled, CoCo also propagates the taint of the message variable to the parameter of the callback function $\Sigma[\Lambda(f)]$. The taint propagation for `sendResponse` and its callback `onResponse` are similar to `sendMessage`. Now we explain the inference rules for "simple one-time requests" one by one.

- `sendMessage`. This rule applies to the `sendMessage` call.
  Condition: The `sendMessage` function is called on the sender end. The two parameters are the message x and the callback function

Figure 2: Operational semantics of CoCo's taint propagation during abstract interpretation

Basic Expressions

$$\frac{\sigma \Rightarrow (\Delta,\Sigma,\Theta), t=([\Phi(src)])}{(\text{get\_taint}(src),\ \sigma) \Rightarrow (\Delta[\Phi(src) \leftarrow t],\Sigma,\Theta)}\ \textsc{Taint Init}$$

$$\frac{\sigma \Rightarrow (\Delta,\Sigma,\Theta), p_{resolved}=\Phi[x[p]/x.const], p_{new}=New(x[p]/x.const)}{(x[p]/x.const,\ \sigma) \Rightarrow \text{if } p_{resolved} != null \text{ then } (\Delta,\Sigma,\Theta) \text{ else } (\Delta[\Phi(p_{new}) \leftarrow \Delta[\Phi(x)]],\Sigma,\Theta)}\ \textsc{Property Access}$$

$$\frac{(x_1,\sigma) \Rightarrow (\Delta_1,\Sigma_1,\Theta_1), (x_2,\sigma) \Rightarrow (\Delta_2,\Sigma_2,\Theta_2), x_{new}=New(x_1 \text{ op } x_2), \Delta'=\Delta[\Phi(x_{new}) \leftarrow \Delta_1[\Phi(x_1)] \cup \Delta_2[\Phi(x_2)]]}{(x_1 \text{ op } x_2,\sigma) \Rightarrow (\Delta',\Sigma_1 \cup \Sigma_2,\Theta_1 \cup \Theta_2)}\ \textsc{Binary Op}$$

$$\frac{\sigma \Rightarrow (\Delta,\Sigma,\Theta), x_{ret}=New(call\ f(x_1,...,x_n)), \Delta'=\Delta[\Phi(x_{ret}) \leftarrow (\bigcup_{i=1}^{n} \Delta[\Phi(x_i) \oplus f])], \Delta''=\Delta[\Phi(x_{ret}) \leftarrow (\bigcup_{i=1}^{n} \Delta[\Phi(x_i)])]}{(call\ f(x_1,...,x_n),\sigma) \Rightarrow \text{if } f \text{ is resolved then } (\Delta,\Sigma,\Theta) \text{ else if } f \in F \text{ then } (\Delta',\Sigma,\Theta) \text{else } (\Delta'',\Sigma,\Theta)}\ \textsc{Func Call}$$

Simple One-Time Requests

$$\frac{\sigma \Rightarrow (\Delta,\Theta,\Sigma), x'=Copy(x), g=\Sigma[\Lambda(f)], \Delta'=\Delta[\Phi(x') \leftarrow \Delta[\Phi(x)]], \Sigma'=\Sigma[\Lambda(r) \leftarrow r]}{(f(x,r),\sigma) \Rightarrow (g(x',S,R),(\Delta',\Theta,\Sigma'))}\ \textsc{sendMessage}$$

$$\frac{\sigma \Rightarrow (\Delta,\Sigma,\Theta), \Sigma'=\Sigma[\Lambda(f) \leftarrow x]}{(f(x),\sigma) \Rightarrow (\Delta,\Sigma',\Theta)}\ \textsc{onMessage-addListener}$$

$$\frac{\sigma \Rightarrow (\Delta,\Theta,\Sigma), x'=Copy(x), g=\Sigma[\Lambda(f)], \Delta'=\Delta[\Phi(x') \leftarrow \Delta\Phi(x)], \Sigma'=\Sigma[\Lambda(f) \leftarrow null]}{(f(x),\sigma) \Rightarrow (g(x'),(\Delta',\Sigma',\Theta))}\ \textsc{onMessage-sendResponse}$$

$$\frac{\sigma, m=New(*)}{(f(x),\sigma) \Rightarrow (\text{get\_taint}(m),x(m,S,R),\sigma)}\ \textsc{onMessageExternal-addListener}$$

$$\frac{\sigma}{(f(x),\sigma) \Rightarrow (\text{set\_sink}(x),\sigma)}\ \textsc{onMessageExternal-sendResponse}$$

Long-Lived Connections

$$\frac{\sigma \Rightarrow (\Delta,\Sigma,\Theta), p=Port(x), g=\Sigma[\Lambda(f)]}{(f(x),\sigma) \Rightarrow (g(p),\sigma)}\ \textsc{connect}$$

$$\frac{\sigma \Rightarrow (\Delta,\Sigma,\Theta), \Sigma'=\Sigma[\Lambda(f) \leftarrow x]}{(f(x),\sigma) \Rightarrow (\Delta,\Sigma',\Theta)}\ \textsc{onConnect-addListener}$$

$$\frac{\sigma \Rightarrow (\Delta,\Sigma,\Theta), x'=Copy(x), \Delta'=\Delta[\Phi(x') \leftarrow \Delta[\Phi(x)]], p=LkupPort(f), g=\Theta[p]}{(f(x),\sigma) \Rightarrow (g(x'),(\Delta',\Sigma,\Theta))}\ \textsc{postMessage}$$

$$\frac{\sigma \Rightarrow (\Delta,\Sigma,\Theta), p=LkupPort(f), \Theta'=\Theta[p \leftarrow x]}{(f(x),\sigma) \Rightarrow (\Delta,\Sigma,\Theta')}\ \textsc{onMessage-addListener}$$

$$\frac{\sigma, p=Port(*)}{(f(x),\sigma) \Rightarrow ((x(p),\sigma))}\ \textsc{onConnectExternal-addListener}$$

$$\frac{\sigma}{(f(x),\sigma) \Rightarrow (\text{set\_sink}(x),\sigma)}\ \textsc{postMessageExternal}$$

$$\frac{\sigma, p=LkupPort(f), m=New(*)}{(f(x),\sigma) \Rightarrow (\text{get\_taint}(m),x(m,p),\sigma)}\ \textsc{onMessageExternal-addListener}$$

r. After sending the message, CoCo copies the message obj x as x' and updates the taint of x to x' by $\Delta[\Phi(x') \leftarrow \Delta[\Phi(x)]]$. The function invoked upon receiving the message is fetched by $g = \Sigma[\Lambda(f)]$. The three parameters for the function g are: x' the copied message, S the Sender variable, and R the built-in response function for simple one-time requests. CoCo also updates the mapping of the "one-time request response" event to the callback function r.

Action: The `sendMessage` function call $f(x, r)$ under state σ is reduced to the call of the function g under the new state.

- `onMessage-addListener`. This rule applies to the message receiver.
  Condition: The `onMessage` function is registered by `addListener` and called on the receiver end upon receiving a message. Its function parameter is the function g in `sendMessage` rule. CoCo updates the callback function mapping of the "one-time request `onMessage`" event, which is mapped by $\Lambda(f)$.
  Action: The function call $f(x, r)$ under state σ can be reduced to the new state with a new mapping of event and callback.
- `onMessage-sendResponse`. This rule applies to the response sent from the receiver.
  Condition: The `sendResponse` function is called from the receiver end inside the `onMessage` callback. The rule is similar to the `sendMessage` rule: CoCo copies the message obj x as x' and updates the taint of x to x' by $\Delta[\Phi(x') \leftarrow \Delta[\Phi(x)]]$. The function invoked upon receiving the response is fetched by $g = \Sigma[\Lambda(f)]$. Note that since this is a one-time request, the event-callback mapping is set to null after the messaging finishes by $\Sigma' = \Sigma[\Lambda(f) \leftarrow null]$.
  Action: The function call $f(x)$ under state σ is reduced to the call of the function g under the new state.
- `onMessageExternal-addListener`. This rule applies to the receipt of external messages.
  Condition: CoCo marks the message m from an external source as tainted.
  Action: The function call under state σ is reduced to getting taint from the wildcard variable and then calls the callback function under state σ. Note that CoCo calls the callback function directly to mimic the external environments.
- `onMessageExternal-sendResponse`. This rule applies to the response for external messages.
  Condition: CoCo treats the message sent to external as a sink.
  Action: The function call $f(x)$ under state σ is reduced to setting the parameter x as sink under the new state.

Third, we describe taint propagation for "long-lived message connections". When a connection is established, CoCo updates Θ to include the port to the corresponding callback function on receiving the message. Then, similar to one-time requests, CoCo copies messages and propagates taints from the sender to the receiver for `port.sendMessage`. Now we explain the inference rules for "long-lived connections" one by one.

- `connect`. This rule applies long-lived message connection.

Condition: This function is called by the party initiating the connection. The parameter x is connection information. The returned object of the `connect` function is a port, which is denoted as $p = Port(x)$ in the rule. g is the function from the party that accepts the connection, fetched by $g = \Sigma[\Lambda(f)]$.
Action: The `connect` function call under state σ is reduced to the g function call under the same state.

- `onConnect-addListener`. This rule applies the callback function for long-lived message connection.
  Condition: This function is called from the party that accepts the connection. CoCo registers the callback function that is invoked when a connection event is fired.
  Action: The function call under state σ is reduced to updating the mapping of event-callback of state σ.
- `postMessage`. This rule applies message posting after a long-lived connection.
  Condition: CoCo copies the message object and updates the taint from the original message object to the copied one. Furthermore, CoCo fetches the message port for the `postMessage` function by $p = LkupPort(f)$ and fetches the corresponding callback of the message port by $g = \Theta[p]$.
  Action: The call of postMessage function under state σ is reduced to the call of function g under state with the updated taint mapping.
- `onMessage-addListener`. This rule applies to the receipt of a message during a long-lived connection.
  Condition: This function is called by the receiver. CoCo first looks up the port for the function f, then registers the callback function on receiving the message by updating the mapping of port p and callback function x by $\Theta' = \Theta[p \leftarrow x]$.
  Action: The function under state σ is reduced to a new state with the updated mapping of message port and callback.
- `onConnectExternal-addListener`. This rule applies the callback function for an external, long-lived message connection.
  Condition: CoCo creates a wildcard port by `Port(*)`.
  Action: The function under state σ is reduced to the calling of callback under the same state.
- `postMessageExternal`. This rule applies to external message posting during a long-lived connection.
  Condition: CoCo treats the external message posting as a sink.
  Then: The `postMessageExternal` function under state σ is reduced to setting the message x as the sink under the same state.
- `onMessageExternal-addListener`. This rule applies to the receipt of an external message during a long-lived connection.
  Condition: CoCo treats the message from the external port as tainted and represents it as a wildcard variable `New(*)`.
  Action: The function under state σ is reduced to getting taint from the wildcard message and calling of callback under the same state.

4.2.2 Vulnerability Detection. CoCo detects extension vulnerabilities via three detailed steps: (i) detection of control- and data-flow patterns, (ii) check on source-sink pair feasibility, and (iii) permission check of "manifest.json".

First, let us start with control- and data-flow patterns. CoCo looks for the following two types of patterns:

- Control-flow patterns. CoCo detects whether there is a control-flow path from an adversary-controlled function to another function with high privileges. Say for example there exists a control-flow path between `chrome.storage.sync.clear()` and message listener. CoCo considers this as a privilege escalation to access browser storage because an adversary can clear the extension's storage.
- Control-flow and taint patterns. CoCo detects whether there exists a taint-flow to a sink's parameter in addition to the aforementioned control-flow path. Take `chrome.tabs.executeScript`, a function call used for executing scripts in the extension context, for example. CoCo detects a vulnerability if the function call is reachable from the control-flow and its parameter is tainted so that an adversary can execute a script.

Second, CoCo checks the feasibility of source-sink pairs, e.g., ensuring that one is from the adversary and the other is from the extension. Specifically, there are two scenarios. (i) If the source contains sensitive data from browser extensions, e.g., a cookie, CoCo checks whether the sink will be accessible to an adversary, e.g., an HTTP request parameter or a message callback. (ii) By contrast, if the source is from an adversary, CoCo checks whether the sink is a sensitive API in the browser extension. After checking the source-sink feasibility, CoCo also checks whether there is source-sink-specific sanitization along the data flow. For example, CoCo checks whether the user is informed of cookie access by inspecting control-flow dependencies like an `if` statement with consent-related APIs such as `window.confirm`: if so, the user consent is considered as a sanitization.

Lastly, CoCo checks the extension's manifest.json to see whether all the involved APIs along the control- and data-flows have the corresponding permission so that the adversary can launch an attack.

4.2.3 Client-side API Modeling. CoCo interacts with a list of modeled, client-side APIs during abstract interpretation, taint propagation, and vulnerability detection. Specifically, we categorize such modeled APIs into two types: event-related and taint-related. Such modeling is done semi-automatically. That is, CoCo has standard function calls for each type, but the determination of API type is decided manually.

First, CoCo maintains an event queue and a dictionary of events and listeners for event-related APIs. When CoCo analyzes an event and its callback function has been registered, CoCo directly creates a thread to analyze the event's callback function. By contrast, if the callback function has not been registered (e.g., for a `sendMessage` event), CoCo adds the event to the event queue. At the same time, CoCo has an event loop, i.e., a special thread, which goes through all the events in the queue: After the event callback is registered, CoCo, particularly the event loop, will analyze the callback via creating a new thread.

Second, CoCo models all the taint-related functions including sources, sinks, and sanitizations. Specifically, CoCo marks corresponding parameter(s) in the source function as tainted, removes such taints from the function call return value for a sanitization function, and reports a vulnerability if the corresponding parameter of a sink function is tainted.

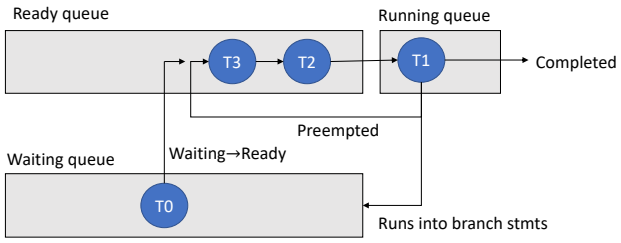

Figure 3: Three queue structures (ready, running, and waiting) used by CoCo and their relations (i.e., how a thread is transferred from one queue to another)

4.3 Thread scheduler

CoCo's thread scheduler is responsible for three tasks: (i) scheduling thread execution, (ii) creating threads to analyze code, and (iii) handling inactive (or debris) threads (i.e., those finish execution). Specifically, CoCo maintains three queue structures, called "Ready", "Running", and "Waiting", as shown in Figure 3 to achieve the goal. The ready queue is a priority queue maintaining all the threads that are ready to be executed and will be fetched onto the running queue. The running queue maintains a list of threads that are currently executed by the thread executor. The waiting queue contains threads that are waiting for other threads' results or that finish execution and are ready to be merged with other threads.

These three queues are connected and threads are moved among three queues based on the aforementioned three tasks. Threads in the ready queue are moved to the running queue by scheduling and back to the ready queue by preemption. Then, threads in the running queue may also be moved to the waiting queue if the execution finishes or encounters a branching statement. Lastly, threads in the waiting queue may be moved back to the ready queue if it is activated. We now describe the three tasks of CoCo in detail.

4.3.1 Scheduling. CoCo schedules threads by moving them from the ready queue to the running queue based on a priority value. Then, CoCo also preempts threads in the running queue and moves them to the ready queue with a new priority value after the allocated time is used up. Below we describe the general scheduling criteria of CoCo and then present a detailed, specific priority calculation method used by CoCo.

Scheduling Criteria. CoCo follows three criteria below to schedule threads.

- *Scheduling Criterion 1* [new code]: The analysis of unseen code has a higher priority than seen code. The first criterion says that CoCo analyzes new code compared with old code that has already been analyzed before. Consider an `if` statement with two branches. The first branch calls a function that is analyzed before, and the second branch calls another unseen function. CoCo prioritizes the analysis of the second branch because this branch has a higher probability of containing a vulnerability.
- *Scheduling Criterion 2* [nested branches]: Given a conditional statement and a branch of the statement, the analysis of a concurrent branch has a higher priority than another embedded conditional statement under this branch. The second criterion says that CoCo analyzes code on the top level compared with the low level for nested conditional statements. The reason is that the top level often contains more high-level semantics compared with the low level where code is often fragmented with details. Say there exists a complex filter in a target program that matches inputs with multiple nested conditional statements. CoCo can quickly skip the analysis of the filter to analyze the rest of the program.
- *Scheduling Criterion 3* [fairness]: Threads that have not been executed for a while have a higher priority than those that are just executed. The third criterion says that CoCo strikes a balance among all the threads when all other conditions are the same. The purpose of this criterion is to prevent starvation and give every thread at least some time to execute.

Priority Calculation. We describe how CoCo calculates the priority in Equation 1 following the three scheduling criteria:

$$P_{\texttt{child}} = P_{\texttt{parent}} + \alpha \cdot \texttt{CovInc}_{\texttt{last}} - \beta \cdot \texttt{brDepth} - \gamma \cdot T_{\texttt{last}} \quad (1)$$

where α, β, and γ are coefficients, $P_{\texttt{parent}}$ the thread's parent's priority, $\texttt{CovInc}_{\texttt{last}}$ the percentage of increased code coverage (Criterion 1) in the last allocated time slot, `brDepth` the branch depth (Criterion 2), and $T_{\texttt{last}}$ the last time (Criterion 3) that the thread is scheduled.

4.3.2 Creation. To start, CoCo creates threads for each component of a browser extension, i.e., content and background scripts. Then, during analysis, CoCo gradually creates more threads for analysis following three creation criteria.

- *Creation Criterion 1* [conditional statement]: CoCo creates a thread for each branch of a conditional statement. CoCo creates new threads for each branch of a conditional statement and also puts the original thread in the waiting queue, which becomes inactive and waits for the finish of branching statements. The reason is that CoCo analyzes later branches immediately without letting them wait for the finish of the beginning branches. Such a creation helps CoCo reach more code as soon as possible.
- *Creation Criterion 2* [event callbacks]: CoCo creates a thread for each event callback function. CoCo creates new threads for each event callback when they are registered. For example, when CoCo encounters `setTimeout`, CoCo creates a new thread for the callback function in its parameter. The procedure is different for message-related events because there are two parties involved. CoCo maintains an event queue to store all the message events sent by `sendMessage`. When the `onMessage` listener is registered, CoCo allocates a special thread that loops through all the messages constantly for handling. Details are described in Section 4.2.3. The reason for such handling of messages is similar: CoCo can quickly reach new code that is embedded as part of event callbacks.
- *Creation Criterion 3* [sequential statements]: CoCo may create threads for sequential statements if they are dataflow independent and the preceding statement is complex to analyze (e.g., the analysis time exceeds a threshold). This is a special, rarely-happened case. Say there are two statements separated by a comma. The first statement takes very long to analyze, e.g., it is an Immediately Invoked Function Expression (IIFE) [3]. CoCo will create a thread for the second statement assuming that there are no data dependencies. Later on, if the first statement modifies an object read by the second statement, CoCo will abort the

execution of the second statement. The reason for Criterion 3 is also to skip complex analysis and reach vulnerability locations.

4.3.3 Thread Activation and Merging. The purpose of activation is to bring threads in the waiting queue back to the ready queue after branching. There are three different policies for such activation to merge different threads.

- [Policy 1] CoCo moves the parent thread from the waiting state to the ready state once one of its created threads terminates.
- [Policy 2] CoCo moves the parent thread from waiting state to ready state when all of its created threads terminate.
- [Policy 3] CoCo creates a copy of the parent thread, and moves the original parent thread from waiting state to ready state once one of its created threads terminates. Then when other branch threads terminate, CoCo moves the copied parent thread to the ready state.

When CoCo moves a parent thread from waiting to ready, CoCo merges the states in the graph-based abstract domains of the debris child thread with the parent. The merge operates on the graph level and generates a new graph based on updates in graphs of child threads. More specifically, the merge starts from the graph of the parent thread and then gradually adds or deletes nodes and edges. On one hand, if an edge or a node is added by any child thread, CoCo will add the same edge or node with the corresponding tag marking the branch. On the other hand, if an edge or a node is deleted by all the threads, CoCo will delete the edge or the node. Otherwise, CoCo will mark the correct branch tag to the edge or the node if one branch still keeps the edge or node.

5 IMPLEMENTATION AND SETUPS

Implementation. Our implementation is open-source with 4,020 lines of code, which is available at this anonymous repo [9]. Our implementation has a pre-processing module, which analyzes manifest.json and extracts all the JavaScript for CoCo to analyze. CoCo will also include modeled client-side APIs (which are shown in Table 1) with each analyzed file. Table 1 shows the sensitive APIs, which are stored as easily-editable text files in CoCo. Our Abstract Syntax Tree (AST) parser is based on an open-source tool, Esprima (https://esprima.org/). Our implementation of the thread executor, specifically the abstract interpretation, is based on an open-source project, ODGen (https://github.com/Song-Li/ODGen), and its representation of Object Dependence Graph as the abstract domain. All the open-source code is excluded from the aforementioned Lines of Code.

Experimental Setups. We describe our datasets and variations of CoCo and state of the art. We prepare two datasets for the evaluation of CoCo: one is unlabelled for zero-day vulnerability detection and the other is labeled mostly by prior works (DoubleX [35] and EmPoWeb [55]). (1) Large-scale extension dataset. Specifically, we crawled 185,076 Chrome extensions from Chrome Web Store in September 2021. We then exclude 20,622 empty extensions, 35 malformed extensions (one that cannot be unzipped, 33 with incorrect manifest.json, and one without manifest.json), and 19,289 themes. (2) Vulnerable extension dataset. This dataset contains 207 vulnerable extensions provided by prior works (DoubleX [35] and EmPoWeb [55]). Specifically, the DoubleX dataset has 184 vulnerable extensions and EmPoWeb has 73 vulnerable extensions. There are 44 overlaps between the two datasets and the combination of the two datasets has 213 extensions containing 256 vulnerabilities. This is because some of these extensions contain more than one vulnerability. We manually verify the exploitability of this combined dataset and find that six vulnerabilities are not exploitable due to a lack of data flows, control flows, or solvable constraints. We thus exclude these six from the dataset.

We now describe different baselines and CoCo variants used in our evaluation. (1) DoubleX [35] is the original implementation from the authors with the newly updated sensitive APIs in Table 1. (2) ODGen-ext [43] is the original, *sequential* implementation plus the newly client-side extension model (e.g., message passing) from CoCo. We use ODGen-ext as another baseline because the original ODGen only supports the analysis of Node.js modules but not browser extensions. (3) CoCo-unguided is a variant of CoCo without coverage guidance. We use CoCo-unguided for the ablation study to understand the importance of coverage guidance.

6 EVALUATION

In this section, we evaluate CoCo and answer five Research Questions (RQs). Our evaluation is on a virtual machine with 128G memory, 20 Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz cores, running Ubuntu 20.04.

6.1 RQ1: Zero-day vulnerabilities

In this research question, we show the capability of CoCo in discovering zero-day vulnerabilities using the large-scale extension dataset with 185,076 extensions. Specifically, our definition of zero-day under the paper's context, following Wikipedia [12], is a browser extension vulnerability that is previously unknown to those who are interested in its mitigation. Therefore, we consider a vulnerability found by CoCo as zero-day if it satisfies the following two criteria: (i) The vulnerability cannot be detected by other tools, particularly neither DoubleX or ODGen-ext. (ii) The vulnerability is not revealed online—e.g., as a bug report, with a CVE identifier, or in another vulnerability dataset—to the best of our knowledge based on an extensive Google search.

In total, CoCo *uniquely* outputs 301 reports from our large-scale extension dataset, which are not reported by either DoubleX or ODGen-ext. Because of the large number, we only manually inspect 50 reports whose corresponding extensions have more than 1,000 users and find 43 zero-day vulnerabilities. A selective list (ranked by # of users) is shown in Table 3. There are five columns in Table 3: extension name, # of users, vulnerability type, status (i.e., whether the vulnerability is reported or confirmed by developers), and the exploit code. Note that CoCo detected vulnerabilities in popular extensions (e.g., Nuance PowerMic Web Extension with more than 200K users). Take Ceibal Library Reader, an extension with 100K+ users, which allows saving the URL of the books that have been downloaded, for example. CoCo finds a vulnerability that allows an adversary to create a new bookmark via sending a message to the extension.

We also show all the zero-day vulnerabilities in Table 4 with a breakdown. Privileged storage access is the most popular one and we further break it down into cookie, bookmark/history and other extension storage. One reason is the large amount of APIs involved

Table 3: A selective list of zero-day vulnerable extensions found by CoCo

Extension Name	# of Users	Vulnerability Type	Status	Exploit Code
Nuance PowerMic Web Extension	200,000+	Code Execution	Reported	`const AttackEvent = new CustomEvent("_nuca_link_request" , {detail: {type: 0x5675, adapterURL: ";console.log(\"attack\")"}}); document.dispatchEvent(AttackEvent);`
Ceibal Library Reader	100,000+	New Bookmark Creation	Reported	`chrome.runtime.sendMessage( "pdjmoegkkmdbohppjcnpfcggajjniepn", {addBookmarkUrl:"http://www.example.com",title:"1"}, function(res) {console.log(res)})`
LinkedIn Email Finder – PIPILEADS	40,000+	Privileged Storage Access	Reported	`window.postMessage({type:"PipiAuthTokenOperation", method:"PipiAuthTokenLogin", token:"123"})`
FixPlay G Server	20,000+	Extension Cookie Access	Reported	`chrome.runtime.sendMessage( "foahoboefnlidmejppiibgnifpbjeknh", {type:"dv", cookie:"hihi"}, function(res){console.log(res)}) chrome.cookies.get({url:"https://drive.google.com", name:""DRIVE_STREAM""},(a)=>{console.log(a)})`
4th Office Edit	1,000+	Privileged AJAX requests	Reported	`document.dispatchEvent(new CustomEvent( "extensions_request", {detail: {domain:"http://www.example.com"}}));`

Table 4: A breakdown by vulnerability type (The list of related sensitive APIs for each type can be found in Table 1.) and exploitable scope (i.e., any websites or those that are defined in the allowlist in the manifest file)

Vulnerability Type	# vulnerabilities = # exploitable by any sites + # exploitable by allowlisted sites
Code Execution	4 = 3 + 1
Privileged AJAX requests	5 = 3+2
Arbitrary File Downloads	1 = 0+1
Privileged Storage Access	33 = 12+21
- Cookie	4 = 1+3
- Bookmark/history	3 = 2+1
- Other storage	26 = 9+17
Total	43 = 18+25

in Table 1. The other is that many extensions may access storage like localStorage. We also break down all the zero-day vulnerabilities by their exploitable scope, i.e., whether they are exploitable by any websites or those that are defined in the allowlist in the manifest file. Overall about 41.9% of websites can be exploited by any websites and the rest needs to be specific sites in the allowlist.

There are two major reasons that CoCo detects zero-day vulnerabilities that are not found by prior works. First, there are many dynamic language features, e.g., bracket syntax and promise, which are not handled by prior works. Here we list an example of one zero-day vulnerability from "Abcd PDF - Chrome New Tab Page" found by CoCo in Listing 2. The vulnerability allows an adversary to obtain the browsing history and bookmarks of the user. Specifically, the exploit code is shown in Lines 2–3. The adversary sends a query at Line 3, which was processed by the content script at Line 5. The content script then sends a message to the background at Line 7, which is received at Line 18. Then, Line 21 calls the `searchHistory` function at Line 27, which calls a sensitive function at Line 29 and then sends the results back at Line 22. Then, the message listener registered by the adversary at Line 2 receives the results. DoubleX cannot detect this vulnerability because of the heavy involvement of `Promise` and `then` function, leading to missing call edges of DoubleX's results. Such dynamic features are often the reason that leads to missed detection of vulnerabilities by prior works such as DoubleX and EmPoWeb. Second, there are vulnerabilities that are triggered by complex inputs and not considered by prior works. "Nuance PowerMic Web Extension" in Table 3 is such an example.

```
1  // exploit code (which retrieves browsing history)
2  window.addEventListener("message", function(msg){console.log(msg
       )})
3  window.postMessage({from:"__newtab", event:"search_history",
       query:""})
4  // content_script.js
5  window.addEventListener('message', function (e) {
6      ...
7      chrome.runtime.sendMessage(e.data, (res) => {
8        const sender = {event: e.data.event,res: res};
9        sender.from = 'ext';
10       window.postMessage(sender, '*');
11     });
12 });
13 // background.js
14 chrome.runtime.onMessage.addListener((request, sender,
       sendResponse) => {
15     listener(request, sender, sendResponse);
16     return true;
17 });
18 function listener(request, sender, sendResponse) {
19     switch(request.event) {
20         case "search_history":
21             searchHistory(request.query).then(res => {
22                 sendResponse(res);
23             });
24             break;
25     }
26 }
27 function searchHistory(query) {
28     return new Promise((resolve, reject) => {
29         chrome.history.search({ text: query, maxResults: 10 }, (
       res) => {
30             resolve(res);
31         });
32     });
33 }
```

Listing 2: A Zero-day Vulnerability Example: Abcd PDF - Chrome New Tab Page

The exploit code of Table 3 shows that the vulnerability is triggered by a custom event listened by the extension as "_nuca_link_request". Such an event is not simulated by DoubleX.

6.2 RQ2: FP and FN

In this research question, we evaluate the false positives and negatives of CoCo and compare with prior works. Figure 4 shows the Venn diagrams of the reported results (including false positives) of all three approaches upon our large-scale extension dataset. First, CoCo detects all the report extensions from ODGen-ext: This is expected because the purpose of our concurrent, coverage-driven abstract interpretation is to increase code coverage and detect more vulnerable extensions. Second, the detection results of CoCo and DoubleX overlap, but each approach has its unique results.

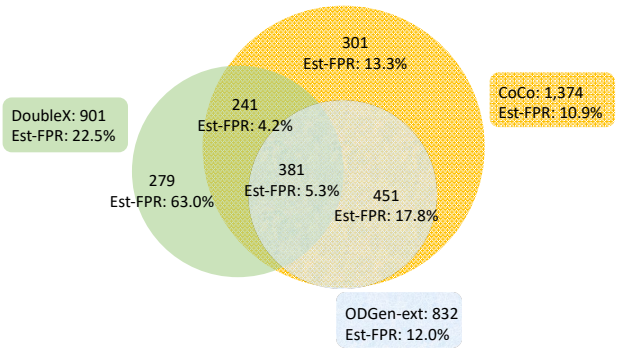

Figure 4: Venn diagram of reported results (including false positives) of CoCo, ODGen-ext, and DoubleX on the large-scale extension dataset. We annotate the estimated False Positive Rate (Est-FPR)—which is obtained manually by randomly sampling 10% reports—under each number.

Table 5: A comparison of false positives (FPs) and negatives (FNs) between CoCo and DoubleX. Note that we sampled 10% of reports of each approach for manual verification of FPs.

Approach	False Positive Rate	False Negative Rate
DoubleX	20/89 (22.5%)	10/250
CoCo	15/137 (10.9%)	2/250
CoCo ∧ DoubleX	3/62 (4.8%)	10/250
¬CoCo ∧ DoubleX	17/27 (63.0%)	250/250
CoCo ∧¬DoubleX	12/75 (16.0%)	242/250
CoCo ∨ DoubleX	32/164 (19.5%)	2/250

Table 6: A comparison of false positives and negatives between CoCo and ODGen-ext

Approach	False Positive Rate	False Negative Rate
ODGen-ext	10/83 (12.0%)	22/250
CoCo	15/137 (10.9%)	2/250
CoCo ∧ ODGen-ext	10/83 (12.0%)	22/250
¬CoCo ∧ ODGen-ext	0/0 (0.0%)	250/250
CoCo ∧¬ODGen-ext	5/54 (9.3%)	230/250
CoCo ∨ ODGen-ext	15/137 (10.9%)	2/250

False Positives. We manually inspect the results from the large-scale extension dataset to evaluate False Positives. Specifically, we define False Positive Rate as FP divided by FP+TP, which indicates the percentage of reported vulnerabilities that are incorrect. That is, we need to spend additional manual efforts to inspect such extensions to filter false positives. Note that our definition is based on exploitability, i.e., we only consider a reported vulnerability TP if it is exploitable.

Due to the large number of vulnerability reports, we manually select 10% of vulnerability reports from each portion (i.e., CoCo, DoubleX, and different combinations of CoCo and DoubleX) and annotate the estimated report FPR in Figure 4. We also break down the Venn diagrams into two comparison pairs and show results in Table 5 (CoCo and DoubleX) and 6 (CoCo and ODGen-ext). Note that ∧ means the detection of both approaches, ¬ means that the followed approach cannot detect the extension, and ∨ means the detection of either approach.

```
// only part of the URL is controlled by the attacker
//background.js
chrome.runtime.onMessageExternal.addListener(
 (request, _, sendResponse) => {
  getMessageDetail(request.user_email, request,sendResponse);
 });
function getMessageDetail(user_email, request,sendResponse){
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "https://gmail.googleapis.com/gmail/v1/users
      /"+user_email+"/messages/"+request.messageId);
    xhr.send(data);
}
```

Listing 3: A false positive case of CoCo: diagrams.net and draw.io Importer

We start with Table 5 and there are several things worth noting here. First, the FPR of CoCo is smaller than that of DoubleX. The reason is that DoubleX often reports a vulnerability while there is no data- or control-flow path between the adversary-controlled input and the sink due to over-approximation. As a comparison, the control-flow produced by CoCo is very accurate due to the adoption of abstract interpretation. At the same time, we also illustrate an FP case of CoCo in Listing 3. CoCo detects this extension as vulnerable because there exists a dataflow from adversary-controlled input to a sink, i.e., a privileged AJAX call. However, we consider this as an FP because the adversary does not have full control over the URL, but only the path. If the adversary provides an invalid path (e.g., email or message ID), the return contents will just be either empty or an error message (like 404).

Next, ¬CoCo ∧ DoubleX has the highest FPR, which is 63.0%. The reason is that the majority of vulnerable extensions are also detected by CoCo. In other words, CoCo also helps DoubleX to reduce FPR when we compare ¬CoCo ∧ DoubleX and CoCo ∧ DoubleX together. Similarly, the reason for FP for DoubleX comes from two perspectives: inaccurate control flow and inaccurate data flow. By contrast, the main reason for CoCo's FP comes from imprecise analysis of AJAX call destinations. Therefore, CoCo helps DoubleX to remove such cases with inaccurate control- and data-flows with abstract interpretation.

We then describe Table 6 and compare CoCo with ODGen-ext. The FPR of CoCo is slightly higher than ODGen-ext and the inclusion of ODGen-ext will slightly increase CoCo's FPR. We are not sure about the exact reasons. It might be that CoCo's unique results are located not as deep as ODGen-ext's results (e.g., not within many embedded branching statements). Therefore, it is easy to exploit such vulnerabilities.

False Negatives. We use the vulnerable extension dataset to evaluate FNs. Specifically, we define False Negative Rate (FNR) as the division of FN over FN+TP, which indicates the percentage of missed vulnerabilities. Table 5 shows the FNR of different combinations of CoCo and DoubleX and in Table 6 the FNR of different combinations of CoCo and ODGen-ext. The false negatives of CoCo are two out of 250 on this dataset. It is worth noting that CoCo detects all the vulnerabilities that DoubleX detects and additionally reports eight more vulnerabilities that cannot be detected by DoubleX. That is why ¬ CoCo ∧ DoubleX cannot detect any vulnerabilities and CoCo ∧ ¬ DoubleX only detects eight.

Another thing worth noting is that CoCo will also sometimes miss the detection of existing vulnerabilities, e.g., those reported by EmpoWeb. One main reason is that some complex built-in functions are not modelled in CoCo, which leads to under-tainting of certain

Table 7: Ablation Study on Concurrency and Coverage-guided Analysis.

Approach	Large-scale dataset	Vulnerable extension dataset
CoCo	1,374	248
CoCo-unguided	1,077	241
ODGen-ext	832	228

objects. Examples of such functions are regular expression and $.Deferred.

6.3 RQ3: Ablation Study

In this research question, we perform an ablation study to understand how each "Co"s contributes to the analysis results of CoCo.

6.3.1 Coverage-guided Analysis. In this part, we show an ablation study of the impacts of coverage-guided analysis by removing coverage guidance from CoCo. Table 7 shows the comparison among CoCo, CoCo-unguided, and ODGen-ext. CoCo-unguided detects fewer vulnerabilities compared with CoCo, but more than ODGen-ext. The reason is that CoCo-unguided often analyzes deep embedded branches with repeated function calls while ignoring some vulnerabilities that exist in the high-level branches.

6.3.2 Concurrency. In part, we show the advantage of concurrency brought by CoCo in the analysis using an ablation study, removing concurrency from CoCo, which essentially becomes ODGen-ext. Specifically, we randomly pick 100 extensions, analyze them and find that 24 of them will time out after ten minutes using ODGen-ext. Then, we compare the code coverage (statement coverage) of these 24 extensions with and without concurrency brought by CoCo. We run each extension extensively with and without concurrency until the code coverage is stable (i.e., staying the same) after ten minutes. Figure 5 shows the Bar Graph of the code coverage improvement comparing CoCo and ODGen-ext. CoCo improves the code coverage by 0% to 14% compared with ODGen-ext with a median value of 4%. This advantage is brought by the concurrent execution of CoCo.

We also show two examples of code coverage of ODGen-ext and CoCo over time in Figure 6 and 7 respectively. First, the code coverage of a ODGen-ext will increase in the beginning but then stay stable after being stuck in analyzing old code. By contrast, the code coverage of a concurrent analysis will increase even if one thread is stuck with analyzing old code. Second, the code coverage of a concurrent analysis is often below 100% even if we give CoCo enough time. One reason is that some extensions may include dead code: For example, if the extension developer includes a JavaScript library, many functions in the library may not be called. This is an advantage: even if there exist vulnerabilities in the dead code, they cannot be exploited by an adversary. Lastly, the code coverage of sequential analysis may be larger than that of concurrent in the beginning as shown in Figure 7. Then, the coverage of concurrent analysis exceeds sequential in the end. The reason is that concurrent analysis has many threads with context switching, which may fall behind sequential analysis in the beginning.

Lastly, we also study how Creation Criteria 2 and 3. Our experiment methodology is as follows. We run CoCo without Creation Criteria 2 or 3 upon our vulnerable extension dataset and compare the number of detected vulnerabilities. Our result shows that Creation Criterion 2 helps the detection of 19 out of 250 vulnerabilities and Creation Criterion 3 the detection of 14 out of 250 vulnerabilities.

6.4 RQ4: Performance

In this research question, we show the performance of CoCo using three metrics: total analysis time, # of threads, and total memory over time. Our methodology is as follows. First, we randomly select 500 extensions from our large-scale dataset and observe the analysis time of three approaches—CoCo, DoubleX and ODGen-ext. Figure 8 shows the Cumulative Distributional Function (CDF) of the total analysis time of three approaches. CoCo is slightly slower than ODGen-ext due to the setup of multiple threads and the time used for switching between threads. Eventually, CoCo manages to finish analyzing more extensions compared with ODGen-ext due to its high code coverage. In the end, the number of finished extensions of CoCo is the same as DoubleX, i.e., there are 32 extensions (6.4%) that time out after ten minutes for both CoCo and DoubleX.

Second, we select five extensions in the aforementioned 500: two with the longest analysis time and the other three randomly from the 500 above. We then show the # of threads and memory overhead over its analysis time. Figure 9 shows the number of threads over analysis time. The number of threads starts to increase when CoCo encounters branching statements, reaches more than 100 for one extension, and then decreases after analyzing branching statements for merging. In another case, the number of threads keeps increasing due to a large number of branching statements in the extension until we kill CoCo after the ten-minute time-out.

Lastly, Figure 10 shows the CDF of the maximum consumed memory in the unit of mebibyte of three approaches, DoubleX, ODGen-ext, and CoCo. The used memory of CoCo is similar to the one used by ODGen-ext, because the main memory consumption is the storage of the graph structure. Both ODGen-ext and CoCo used less memory compared with DoubleX for more than 95% of extensions because of different representations of the program dependency graph and object dependence graph. ODGen-ext and CoCo used more memory for the rest of 5% of extensions because abstract interpretation will generate more nodes for some particular program structures like recursion.

7 DISCUSSION

Responsible Disclosure. We follow standard responsible disclosure procedures to inform vulnerable extension developers and give them 45 days to fix the vulnerability before public release. That said, we wrote emails to all the developers of the zero-day vulnerabilities belonging to 84 extensions when we manually verified them as exploitable regardless of whether the vulnerability is detected by CoCo, DoubleX, or ODGen-ext. More specifically, we only find the developers' contact information of 39 out of 43 vulnerabilities that are uniquely detected by CoCo as vulnerable, and make corresponding reports to the developers of the 37 affected extensions. In addition, we also find contact information of zero-day vulnerabilities that are detected by either DoubleX or ODGen-ext as well and confirmed as exploitable. Thus, we make corresponding reports to those developers of affected extensions as well. Note that many of these vulnerabilities reported by either DoubleX or ODGen-ext are

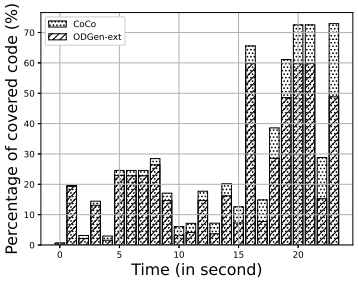

Figure 5: Code Coverage Comparison of ODGen-ext and CoCo over 24 Timed-out Extensions

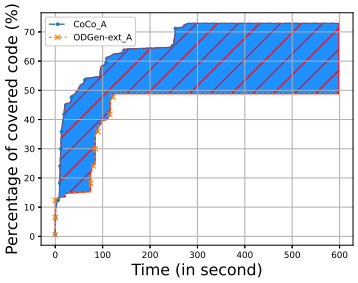

Figure 6: Code Coverage Increase of CoCo vs. ODGen-ext over Time of Zuora RBM Connect Plugin

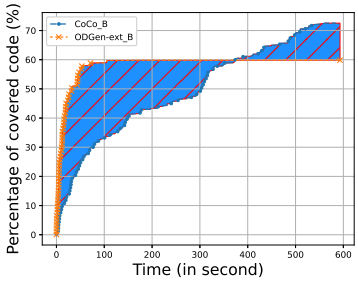

Figure 7: Code Coverage Increase over Time of CoCo vs. ODGen-ext for a specific extension [7]

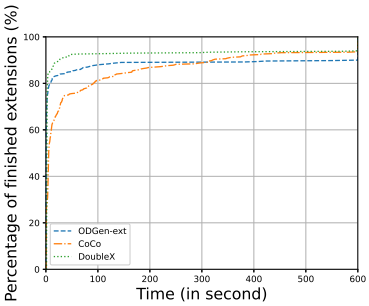

Figure 8: CDF of Total Analysis Time for 500 Random Extensions

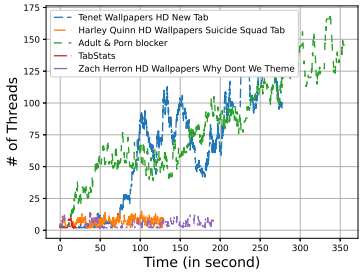

Figure 9: # of Threads vs. Analysis Time of CoCo

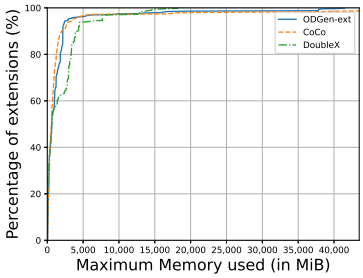

Figure 10: CDF of Maximum Memory for 500 Random Extensions

also detected by CoCo; that is, only four vulnerabilities belonging to four extensions are only detected by DoubleX but not CoCo. In the email, we describe where the vulnerability locates and how we suggest patching the vulnerability. So far we have only received one confirmation and unfortunately, none of the vulnerable extensions have been fixed yet. We are still working with developers as well as extension marketplace operators for fixes.

Manifest V3 Extensions. Recently, Google Chrome releases Manifest V3, which makes two major changes beyond the manifest file: (i) migrating background scripts to service workers, (ii) adding additional constraints for cross-origin requests in the content scripts. CoCo is *compatible* with Manifest V3 extensions with three major changes. First, CoCo supports a special parsing component for all V3 manifest files. Second, CoCo creates a special thread for service workers just like background scripts. Here are the analysis results. So far, there are 3,798 extensions in our large-scale extension dataset using Manifest V3. CoCogenerates 21 reports as potentially vulnerable and our further manual verification shows that 19 out of 21 are true positives.

Constraint Solving. CoCo statically finds a data path between an adversary-controlled input and a sink function. The found path may not be feasible due to certain control- or data-flow constraints. This is the same as the state-of-the-art approach, particularly DoubleX. Furthermore, false positives caused by a lack of constraint solving is relatively small based on our manual checking. We leave it as our future work to include constraint solvers in CoCo.

Soundness. CoCo is the same as all previous static analysis of JavaScript, which is not sound. Specifically, we describe soundness conditions below. First, dynamic object creation in CoCo is sound only if all the related variables can be precisely resolved. That is, if one variable is related to user inputs, CoCo makes an overapproximation. Second, dynamic generation of code in CoCo is sound only if the generated code value is resolvable. That is, if the code generation parameter is related to user inputs, CoCo will skip the analysis. Note that since dynamic code generation is also a sink for code execution, the implementation choice does not affect vulnerability analysis.

8 RELATED WORK

Browser Extension Security. In 2007, Louw et al. [58] studied the security issues of browser extensions and introduced an integrity-checking mechanism to control the extensions' installation and loading process. In 2010, VEX [14] applied static information-flow analysis to the browser extensions and used static flow patterns and unsafe programming practices to highlight potential security vulnerabilities. Later, IBEX [36] adopted Datalog and data flow policies to limit the API usage of browser extensions. Then, Carlini et al. [26] evaluated the effectiveness of Chrome extension security architecture by a security review of 100 Chrome extensions, and Wang et al. [59] did a measurement study to analyze extension behaviors. In 2013, Sentinel [47] introduced a user-controllable policy enforcer for the Firefox browser that gives fine-grained control to the JavaScript Firefox extensions. In 2014 and 2015, multiple researchers focused on the detection of malicious extensions and vulnerabilities. Hulk [40] leveraged extension-related dynamic pages and employed a fuzzer to detect malicious extensions. Calzavara et al. [16] proposed a formal security analysis of browser extensions to show that message-passing APIs may lead to privilege escalation attacks. WebEval [37] adopted dynamic analysis, static analysis, and reputation tracking to detect malicious extensions. Onarlioglu

et al. [48] investigated the security issues of the Firefox XPCOM extension APIs and introduced methods to detect related vulnerabilities. In 2016, researchers used static and dynamic analysis to detect vulnerabilities and sensitive information leakage. Anil et al. [54] extended the colluding attacks to the extension domain and showed that the collusion between two extensions may lead to private information leakage. CrossFire [15] adopted a multi-stage lightweight static analyzer to detect instances of extension-reuse vulnerabilities on top of Firefox. ExtensionGuard [27] used a customizable dynamic taint tracker to mark the sensitive information, and then detect information leakage during runtime.

In 2017 and 2018, there were some works studying privacy issues of browser extensions. Starov et al. [56] reported a large-scale study of privacy leakage enabled by extensions, and Ex-Ray [60] presented a dynamic technique that was based on the network traffic patterns for identifying privacy-violating extensions. Aggarwal et al. [13] detected and defended spying extensions by using RNN with the sequence of browser API calls. Mystique [28] used static analysis to obtain the data-flow and control-flow graphs and modifies Chromium to detect the leakage of private information. In 2019 and 2020, EmPoWeb [55] used call graph analysis to investigate how communication-related APIs can influence the security of browser extensions. Pantelaios et al. [51] detected malicious browser extensions through their update deltas and did a large-scale to-date measurement study. Recently in 2021, DoubleX [35] analyzed taint flows to detect browser extension vulnerabilities without the support of dynamic features. In 2022, Benjamin et al. [31] presented a systematic study of attack entry points in the browser extension ecosystem.

Note that prior works on browser extension vulnerability detection either adopted dynamic analysis [16, 37], used policy enforcers [47, 48], or only had limited support for dynamic features [35].

General Web Security. General web security [17–20, 22–25, 49, 50, 61, 62] has been studied for many years. We start with static analysis. Jensen et al. [38] use static analysis to detect type-related and dataflow-related programming errors of client-side JavaScript applications that interact with the HTML, DOM, and browser APIs. HideNoSeek [32], JShield [21], JaSt [34], and JSTap [33] adopt static analysis to detect malicious client-side JavaScript applications. JSIsolate [64] provides an isolated and reliable JavaScript execution environment based on the dependency relationship of different JavaScript program components. JAW [41] detects client-side CSRF vulnerabilities by modeling browser objects in the Hybrid Property Graphs. As for dynamic analysis. Deemon [52] combines dynamic analysis and property graphs to detect the CSRF vulnerability. Melicher et al. [45] and Steffens et al. [57] adopt dynamic analysis to detect DOM-based XSS vulnerabilities. JSObserver [63] focuses on the code integrity problem of client-side JavaScript that is caused by global identifier conflicts. Black Widow [30], a black box data-driven approach to web crawling and scanning, finds more cross-site scripting vulnerabilities with no false positives.

9 CONCLUSION

In this paper, we design and implement a new framework, called CoCo, to parallelize abstract interpretation for analyzing browser extensions with concurrent taint analysis. Specifically, CoCo creates concurrent analysis for new branches and events and propagates taints across different threads. Then, CoCo schedules analysis to prioritize code coverage so that it can always try to reach new code and find vulnerabilities. We evaluate CoCo using both ground truth and real-world extension datasets and compare CoCo with the state-of-the-art approach, DoubleX, as well as a modified version of ODGen. Our evaluation shows that CoCo detects zero-day vulnerabilities that cannot be detected by state-of-the-art approaches.

ACKNOWLEDGEMENT

We would like to thank anonymous shepherd and reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361 and Defense Advanced Research Projects Agency (DARPA) under AFRL Definitive Contract FA875019C0006 and a DARPA Young Faculty Award (YFA) under Grant Agreement D22AP00137-00 as well as an Amazon Research Award (ARA) 2021 and a Visa Research Award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, DARPA, Amazon, or Visa.

REFERENCES

[1] [n.d.]. *Architecture overview.* https://developer.chrome.com/docs/extensions/mv3/architecture-overview/.
[2] [n.d.]. *Changes to Cross-Origin Requests in Chrome Extension Content Scripts.* https://www.chromium.org/Home/chromium-security/extension-content-script-fetches/.
[3] [n.d.]. *Definition of IIFE.* https://developer.mozilla.org/en-US/docs/Glossary/IIFE.
[4] [n.d.]. *FromDocToPdf: exposes browsing history to all websites.* https://bugs.chromium.org/p/project-zero/issues/detail?id=1557.
[5] [n.d.]. Google Scholar Button. https://chrome.google.com/webstore/detail/google-scholar-button/ldipcbpaocekfooobnbcddclnhejkcpn. [Online; Accessed on 07-June-2022].
[6] [n.d.]. Grammarly: Grammar Checker and Writing App. https://chrome.google.com/webstore/detail/grammarly-grammar-checker/kbfnbcaeplbcioakkpcpgfkobkghlhen. [Online; Accessed on 07-June-2022].
[7] [n.d.]. *Kino No Tabi Backgrounds HD S Journey New Tab.* https://chrome-stats.com/d/agpijpbfbjfdahhjjigjbhfeogijlajm.
[8] [n.d.]. *Message passing.* https://developer.chrome.com/docs/extensions/mv3/messaging/.
[9] [n.d.]. *Open-source Repository.* https://github.com/CoCoAbstractInterpretation/CoCo.git.
[10] [n.d.]. Operational semantics (2022) Wikipedia. https://en.wikipedia.org/wiki/Operational_semantics. [Online; Accessed on March 21, 2023].
[11] [n.d.]. *Video Downloader Extension: Universal XSS.* https://bugs.chromium.org/p/project-zero/issues/detail?id=1555.
[12] [n.d.]. *[Wikipedia] Zero-day (computing).* https://en.wikipedia.org/wiki/Zero-day_(computing).
[13] Anupama Aggarwal, Bimal Viswanath, Liang Zhang, Saravana Kumar, Ayush Shah, and Ponnurangam Kumaraguru. 2018. I Spy with My Little Eye: Analysis and Detection of Spying Browser Extensions. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 47–61. https://doi.org/10.1109/EuroSP.2018.00012
[14] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. 2010. VEX: Vetting Browser Extensions for Security Vulnerabilities. In *19th USENIX Security Symposium (USENIX Security 10)*. Washington, DC.
[15] Ahmet Salih Buyukkayhan, Kaan Onarlioglu, William K. Robertson, and Engin Kirda. 2016. CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities. In *NDSS 2016*.
[16] Stefano Calzavara, Michele Bugliesi, Silvia Crafa, and Enrico Steffinlongo. 2015. Fine-Grained Detection of Privilege Escalation Attacks on Browser Extensions. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 510–534.
[17] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. 2017. Deterministic Browser. In *CCS* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 163–178. https://doi.org/10.1145/3133956.3133996

[18] Yinzhi Cao, Song Li, Erik Wijmans, et al. 2017. (Cross-) Browser Fingerprinting via OS and Hardware Level Features.. In *NDSS*.

[19] Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, and Yan Chen. 2010. Virtual Browser: A Web-Level Sandbox to Secure Third-Party JavaScript without Sacrificing Functionality. In *CCS (CCS '10)*. Association for Computing Machinery.

[20] Yinzhi Cao, Xiang Pan, Yan Chen, and Jianwei Zhuge. 2014. JShield: Towards Real-Time and Vulnerability-Based Detection of Polluted Drive-by Download Attacks. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. Association for Computing Machinery, New York, NY, USA.

[21] Yinzhi Cao, Xiang Pan, Yan Chen, and Jianwei Zhuge. 2014. JShield: towards real-time and vulnerability-based detection of polluted drive-by download attacks. *Proceedings of the 30th Annual Computer Security Applications Conference* (2014).

[22] Yinzhi Cao, Xiang Pan, Yan Chen, Jianwei Zhuge, Xiaobin Qian, and Jian Fu. 2015. Malicious code detection technologies. US Patent 9,213,839.

[23] Yinzhi Cao, Vaibhav Rastogi, Zhichun Li, Yan Chen, and Alexander Moshchuk. 2013. Redefining web browser principals with a Configurable Origin Policy. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 1–12. https://doi.org/10.1109/DSN.2013.6575317

[24] Yinzhi Cao, Yan Shoshitaishvili, Kevin Borgolte, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2014. Protecting Web-Based Single Sign-on Protocols against Relying Party Impersonation Attacks through a Dedicated Bi-directional Authenticated Secure Channel. In *Research in Attacks, Intrusions and Defenses*.

[25] Yinzhi Cao, Vinod Yegneswaran, and Yan Chen. 2012. PathCutter: Severing the Self-Propagation Path of XSS JavaScript Worms in Social Web Networks.. In *NDSS*.

[26] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. 2012. An Evaluation of the Google Chrome Extension Security Architecture. In *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA, 97–111.

[27] Wentao Chang and Songqing Chen. 2016. ExtensionGuard: Towards runtime browser extension information leakage detection. In *2016 IEEE Conference on Communications and Network Security (CNS)*. 154–162.

[28] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[29] Patrick Cousot. 1996. Abstract interpretation. *ACM Computing Surveys (CSUR)* 28, 2 (1996), 324–328.

[30] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black Widow: Blackbox Data-driven Web Scanning. In *2021 IEEE Symposium on Security and Privacy (SP)*.

[31] Benjamin Eriksson, Pablo Picazo-Sanchez, and Andrei Sabelfeld. 2022. Hardening the Security Analysis of Browser Extensions. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22)*.

[32] Aurore Fass, Michael Backes, and Ben Stock. 2019. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery.

[33] Aurore Fass, Michael Backes, and Ben Stock. 2019. JStap: A Static Pre-Filter for Malicious JavaScript Detection. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, 257–269.

[34] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. 2018. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

[35] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale, In ACM CCS 2021. *ACM CCS*.

[36] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. 2011. Verified Security for Browser Extensions. In *2011 IEEE Symposium on Security and Privacy*. 115–130. https://doi.org/10.1109/SP.2011.36

[37] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. 2015. Trends and Lessons from Three Years Fighting Malicious Extensions. In *24th USENIX Security Symposium*.

[38] Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[39] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS) (LNCS, Vol. 5673)*. Springer-Verlag.

[40] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *23rd USENIX Security Symposium*.

[41] Soheil Khodayari and Giancarlo Pellegrino. 2021. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In *30th USENIX Security Symposium (USENIX Security 21)*. 2525–2542.

[42] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting Node.js Prototype Pollution Vulnerabilities via Object Lookup Analysis. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[43] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA.

[44] Laurent Mauborgne and Xavier Rival. 2005. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming*. Springer, 5–20.

[45] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *Network and Distributed System Security Symposium (NDSS)*.

[46] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. Nodest: Feedback-Driven Static Analysis of Node.Js Applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 455–465.

[47] Kaan Onarlioglu, Mustafa Battal, William Robertson, and Engin Kirda. 2013. Securing Legacy Firefox Extensions with SENTINEL. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Konrad Rieck, Patrick Stewin, and Jean-Pierre Seifert (Eds.).

[48] Kaan Onarlioglu, Ahmet Salih Buyukkayhan, William Robertson, and Engin Kirda. 2015. SENTINEL: Securing Legacy Firefox Extensions. *Computers & Security* 49 (2015), 147–161. https://doi.org/10.1016/j.cose.2014.12.002

[49] Xiang Pan, Yinzhi Cao, and Yan Chen. 2015. I do not know what you visited last summer: Protecting users from third-party web tracking with trackingfree browser. In *NDSS*.

[50] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. 2016. CSPAutoGen: Black-Box Enforcement of Content Security Policy upon Real-World Websites. In *CCS 2016 (CCS '16)*.

[51] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. 2020. *You've Changed: Detecting Malicious Browser Extensions through Their Update Deltas*. Association for Computing Machinery, New York, NY, USA, 477–491.

[52] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. 2017. Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*.

[53] Xavier Rival and Laurent Mauborgne. 2007. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 5 (2007), 26–es.

[54] Anil Saini, Manoj Singh Gaur, Vijay Laxmi, and Mauro Conti. 2016. Colluding browser extension attack on user privacy and its implication for web browsers. *Computers & Security* 63 (2016), 14–28.

[55] Dolière Francis Somé. 2019. EmPoWeb: Empowering Web Applications with Browser Extensions. In *IEEE Security and Privacy Symposium*.

[56] Oleksii Starov and Nick Nikiforakis. 2017. Extended Tracking Powers: Measuring the Privacy Diffusion Enabled by Browser Extensions. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*.

[57] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *NDSS*.

[58] Mike Ter Louw, Jin Soon Lim, and V. N. Venkatakrishnan. 2007. Extensible Web Browser Security. In *Proceedings of the 4th DIMVA (2007)*.

[59] Jiangang Wang, Xiaohong Li, Xuhui Liu, Xinshu Dong, Junjie Wang, Zhenkai Liang, and Zhiyong Feng. 2012. An Empirical Study of Dangerous Behaviors in Firefox Extensions. In *Information Security*.

[60] Michael Weissbacher, Enrico Mariconti, Guillermo Suarez-Tangil, Gianluca Stringhini, William Robertson, and Engin Kirda. 2017. Ex-Ray: Detection of History-Leaking Browser Extensions. In *ACSAC 2017*.

[61] Shujiang Wu, Song Li, Yinzhi Cao, and Ningfei Wang. 2019. Rendered Private: Making GLSL Execution Uniform to Prevent WebGL-based Browser Fingerprinting.. In *USENIX Security*.

[62] Shujiang Wu, Pengfei Sun, Yao Zhao, and Yinzhi Cao. 2023. Him of Many Faces: Characterizing Billion-scale Adversarial and Benign Browser Fingerprints on Commercial Websites. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society.

[63] Mingxue Zhang and Wei Meng. 2020. Detecting and Understanding JavaScript Global Identifier Conflicts on the Web. In *Proceedings of ESEC/FSE 2020* (Virtual Event, USA). 38–49.

[64] Mingxue Zhang and Wei Meng. 2021. JSISOLATE: Lightweight in-Browser JavaScript Isolation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.