DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale

Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock
CISPA Helmholtz Center for Information Security
{aurore.fass,doliere.some,backes,stock}@cispa.de

Abstract

Browser extensions are popular to enhance users' browsing experience. By design, they have access to security- and privacy-critical APIs to perform tasks that web applications cannot traditionally do. Even though web pages and extensions are isolated, they can communicate through messages. Specifically, a vulnerable extension can receive messages from another extension or web page, under the control of an attacker. Thus, these communication channels are a way for a malicious actor to elevate their privileges to the capabilities of an extension, which can lead to, e.g., universal cross-site scripting or sensitive user data exfiltration. To automatically detect such security and privacy threats in benign-but-buggy extensions, we propose our static analyzer DoubleX. DoubleX defines an Extension Dependence Graph (EDG), which abstracts extension code with control and data flows, pointer analysis, and models the message interactions within and outside of an extension. This way, we can leverage this graph to track and detect suspicious data flows between external actors and sensitive APIs in browser extensions.

We evaluated DoubleX on 154,484 Chrome extensions, where it flags 278 extensions as having a suspicious data flow. Overall, we could verify that 89% of these flows can be influenced by external actors (i.e., an attacker). Based on our threat model, we subsequently demonstrate exploitability for 184 extensions. Finally, we evaluated DoubleX on a labeled vulnerable extension set, where it accurately detects almost 93% of known flaws.

CCS Concepts

• **Security and privacy → Web application security**; **Browser security**.

Keywords

Web Security, Browser Extension, JavaScript, Vulnerability Detection, Static Analysis, Data Flow Analysis

1 Introduction

The Web has become a popular ecosystem used by billions of people every day. To extend their browser functionality, users install browser extensions. For the most popular desktop browser Chrome (with a market share of 67% [78]), there are almost 200,000 extensions, totaling over 1.2 billion installs [26]. While some extensions merely customize user browser interface, others serve more security- and privacy-critical tasks, e.g., to be effective, an ad-blocker needs to modify web page content or intercept network requests. To achieve this, and contrary to regular JavaScript in web pages, extensions have privileged capabilities, such as downloading arbitrary files or accessing arbitrary cross-domain data.

Given their elevated privileges, extensions attract the interest of attackers [1, 35, 39, 81, 82]. Still, Google engineers are actively working on detecting such malicious extensions in their store. In February 2020, they removed 500 extensions that were exfiltrating user data [41]; in April 2020, 49 additional extensions that were hijacking users' cryptocurrency wallets [42]; in June 2020, an extra 70 spying extensions [48]; and, in December 2020, extensions redirecting to phishing websites [43]. In addition, before being published, extensions are reviewed by Chrome's vetting system to flag extensions requiring many or powerful privileges for further analyses [16] and to directly detect the ones that may contain or spread malicious software. This process makes it harder to have malicious extensions in the store today.

Still, malicious extensions represent only a fraction of the extensions that may lead to security or privacy issues. In fact, an attacker can also abuse vulnerable extensions to elevate its privileges through the capabilities of an extension. To this end, an attacker can leverage an extension's communication channels to send payloads to this extension, tailored to exploit its vulnerabilities. Such vulnerabilities can lead to, e.g., universal cross-site scripting (XSS) (i.e., the ability to execute code in every website even without a vulnerability in the site itself) or sensitive user data exfiltration. Due to their inherently benign intent, these vulnerable extensions are more challenging to detect than malicious ones, e.g., as they are not doing anything suspicious. Furthermore, even though they do require powerful privileges, their benign nature allows them to pass the review process. While some previous works have focussed on vulnerable extensions, they were either purely formal [7], specific to the deprecated Firefox XPCOM [52] infrastructure [5, 6], or based on primarily manual analysis [8]. To the best of our knowledge, only EmPoWeb from Somé [72] focuses on analyzing extensions' susceptibility to attacks through external messages at scale. In practice, though, his work is based on a lightweight call graph analysis and yields an extremely high number of reported extensions to manually vet: of the 66k Chrome extensions analyzed, it flagged 3.3k as suspicious, and 95% of them were false alarms. In fact, we currently lack a precise analyzer to perform an analysis at scale and limit the number of extensions falsely reported as vulnerable so as to cut down the manual effort.

In this paper, we introduce our static analyzer DoubleX, which relies on an advanced data flow analysis to accurately track data from and toward security- and privacy-critical APIs in extensions. In particular, due to its entirely static character, DoubleX has complete coverage of the available code. Specifically, we propose a semantic abstraction of extension code, including control and data flows, and pointer analysis. In addition, DoubleX models interactions between extension components with a *message flow*. We refer to the resulting structure as the *Extension Dependence Graph* (EDG). DoubleX then leverages the EDG to detect external messages coming from web pages or extensions and flags them as attacker-controllable. At the same time, it collects security- and privacy-critical APIs in browser extensions. Finally, DoubleX performs a data flow analysis to identify any path between external actors (i.e., an attacker) and these sensitive APIs. DoubleX summarizes its findings in fine-grained data flow reports.

We analyzed 154,484 Chrome extensions, 278 of which we flagged as having externally controllable data flows or exfiltrating sensitive user information. For those, we could verify that 89% of the data flows can be influenced by an attacker, which highlights DoubleX precision. In addition, we detected 184 extensions (with 209 vulnerabilities) that are exploitable under our threat model, leading to, e.g., arbitrary code execution in any website. Also, we evaluated DoubleX recall on a ground-truth vulnerable extension set, where it accurately flags 92.64% of known vulnerabilities. Finally, DoubleX is fast and can analyze 93% of our extension set in less than 20s per extension, with a median time of 2.5s, highlighting its practicability for an accurate analysis at scale.

To sum up, our paper makes the following contributions:

- We introduce our static analyzer DoubleX to analyze browser extensions at scale. Specifically, we define an Extension Dependence Graph (EDG), which provides a semantic abstraction of extension code (including control and data flows, and pointer analysis) and models message interactions within and outside of an extension.
- We leverage the EDG to perform a data flow analysis to track data from and toward security- and privacy-critical APIs in browser extensions.
- We perform a large-scale analysis of Chrome extensions and identify 184 vulnerable extensions. In addition, we highlight DoubleX precision (89% verified dangerous data flows) and recall (detection of 92.64% of known vulnerabilities).
- For reproducibility, follow-up work, and practical detection of suspicious external data flows in specific APIs of browser extensions, we make DoubleX publicly available [27].

2 Browser Extensions

Browser extensions are third-party programs, which users can install to extend browser functionality, e.g., by adding ad-blocking capabilities or better integration with shopping sites. In this section, we first present the extension architecture with a highlight on security mechanisms extensions implement. Then, we focus on the message-passing APIs they use to communicate.

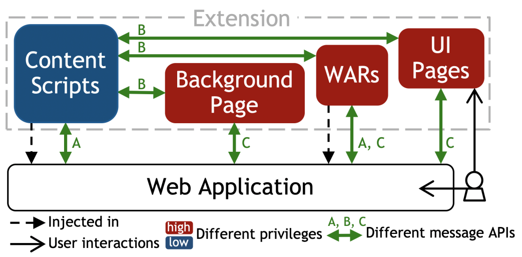

Figure 1: Extension architecture

2.1 Architecture and Security Considerations

By design, extensions have access to privileged APIs and features. Contrary to JavaScript in web pages, which is restricted by the Same-Origin Policy (SOP) [59], extensions can access arbitrary cross-domain in the logged-in context of the user's browser and inject code into any document. Due to their elevated privileges, extensions may introduce security and privacy threats and put their user base at risk. To limit those risks, extensions only have access to the permissions explicitly declared in their `manifest.json` [14, 57]. Such permissions include the possibility for an extension to read/write user data on any or specific web pages (`host`), to store/retrieve data from the extension storage (`storage`), to download arbitrary files (`downloads`), or to access users' browsing history (`history`).

As represented in Figure 1, an extension is divided into four main components. The core logic of an extension is implemented through a background page (or background scripts),[1] which runs independently of the lifetime of any particular web page or browser window. Through the background page, or if defined in the manifest, an extension can inject content scripts to run along with web applications. These content scripts can use the standard DOM APIs to read and modify web pages and have access to `localStorage`, similarly to the scripts loaded by web pages. UI pages enable users to customize an extension's behavior, e.g., over different options, settings, or pop-ups. Finally, through definition in the manifest, an extension can expose Web Accessible Resources (WARs), e.g., scripts to be executed on every page. While the highly privileged background page, UI pages, and WARs have access to the full extension's capabilities, the less privileged content scripts only have access to the `host`[2] and `storage` permissions.

2.2 Message Passing in Extensions

To communicate with web pages and other extensions, an extension relies on message passing. In this section, we present the communication channels between a web page and each extension component, within an extension, and between two extensions. Figure 2 shows an overview of the message-passing APIs.

Web Page - Content Script — Web pages and content scripts communicate over regular postMessages [62] (similarly to the communication between two web pages). Likewise, to receive messages, they use the `addEventListener` or `onmessage` API [55, 61], as shown

[1] Replaced with service workers, for Chrome Manifest V3 [19]
[2] Starting with Chrome Manifest V3, content scripts are subject to the same request rules as the page they are running within [23]

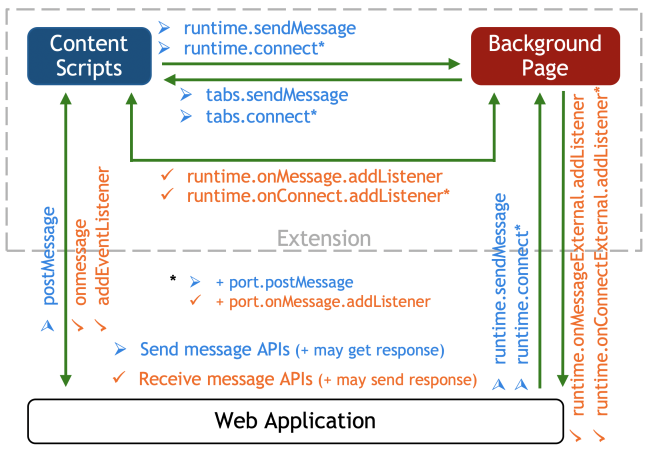
Content Scripts
runtime.sendMessage
runtime.connect*
Background Page
tabs.sendMessage
tabs.connect*
runtime.onMessage.addListener
runtime.onConnect.addListener*
Extension
postMessage
onmessage
addEventListener
* + port.postMessage
+ port.onMessage.addListener
Send message APIs (+ may get response)
Receive message APIs (+ may send response)
runtime.sendMessage
runtime.connect*
runtime.onMessageExternal.addListener
runtime.onConnectExternal.addListener*
Web Application

Figure 2: Extension message-passing APIs

```
// Web page code
window.postMessage("Hi CS", "*"); // Sends

// Content script code
window.onmessage = function(event) {
 received = event.data; // received = "Hi CS"
}
```

Listing 1: Messages: web page - content script

```
// Content script code
chrome.runtime.sendMessage("Hi BP", function(response) {
 csReceived = response.farewell; // csReceived = "Bye CS"
});

// Background page code
chrome.runtime.onMessage.addListener(
 function(request, sender, sendResponse) {
  bpReceived = request; // bpReceived = "Hi BP"
  sendResponse({farewell: "Bye CS"});
});
```

Listing 2: Messages: content script - background page

in Listing 1 (orange and blue refer to receiving and sending messages, respectively). By default, content scripts receive all messages sent toward the window in which they are injected. Thus, if a web page running a content script receives a `postMessage` from another page, the content script's handler is also invoked.

Content Script - Background Page — There are two types of APIs to exchange messages between a content script and a background page. The *one-time requests* API aims at sending a single message and receiving a response. In contrast, *long-lived connections* leverage an established message port and stay open to exchange multiple messages [18]. As shown in Figure 2, the content script uses `runtime.sendMessage` (one-time) or `runtime.connect` (long-lived) to send messages.[3] Similarly, the background page sends requests with `tabs.sendMessage` (one-time) or `tabs.connect` (long-lived). In both cases, the last parameter of these APIs can be a callback to access the response sent by the other component (for Firefox, these APIs can return a `Promise` [58] instead of invoking a callback). As for receiving messages (and responding), both components register a listener: `runtime.onMessage.addListener` (one-time) or `runtime.onConnect.addListener` (long-lived). Listing 2 illustrates Chrome one-time requests.

Web Page - Background Page — For Chromium-based extensions, a web application and a background page can directly communicate under two assumptions [18]. First, the extension should fill the `externally_connectable` field in its manifest with specific URLs, to allow the communication with the corresponding web pages only. Second, the communication can only be initiated by the web application. As presented in Figure 2, the web application sends requests (and gets a response) with `runtime.sendMessage` (one-time) or `runtime.connect` (long-lived). The background page receives messages (and responds) with `runtime.onMessageExternal.addListener` (one-time) or `runtime.onConnectExternal.addListener` (long-lived). We give an example in Listing 6 in the Appendix.

[3]For legibility reasons, we omit `browser`/`chrome` from the APIs, which would be, e.g., `chrome.runtime.sendMessage` or `browser.runtime.sendMessage`

Case of UI Pages and WARs — Like the background page, UI pages and WARs are part of the extension core. To exchange messages with the content scripts or with a web page, they use the same APIs as those used by the background page, respectively. As WARs can be injected as iframes in web pages, WARs also leverage the same APIs as the content scripts to interact with a web page. Figure 1 summarizes the three sorts of messages used by the extension components, while Figure 2 presents the specific APIs.

Extension A - Extension B — Finally, two extensions can communicate. In this case, the message-passing APIs are the same as those for the communication between a background page and a web application. Still, contrary to the communication with a web page, communication is enabled *by default* with all extensions [15]. To interact with specific extensions only, an extension must explicitly declare the IDs of allowed extensions in its manifest.

3 Threat Model

Browser extensions can interact with web pages and other extensions. By design, malicious actors can send specific messages to a vulnerable extension, tailored to exploit its flaws. Given extensions' elevated privileges, attackers could gain the following capabilities:

- **Code execution**: attackers can execute arbitrary code in the extension (or content script) context. For example, through `eval`, they could exploit all the extension's permissions. Through `tabs.executeScript`, they may gain a universal XSS, i.e., execute arbitrary code in every web page even without a vulnerability in the page itself;
- **Triggering downloads**: they can download and save arbitrary files on users' machines without prior notice;
- **Cross-origin requests**: they can bypass the Same-Origin Policy;
- **Data exfiltration**: they can access sensitive user information such as cookies, browsing history, or most visited websites, leading to, e.g., session hijacking, tracking, or fingerprinting.

In this paper, we focus on two attacker scenarios: a *Web Attacker* and an attacker abusing a *Confused Deputy* through an unprivileged extension. In the first scenario, an attacker can trick a user into visiting a web page that can communicate with an extension. This page, which can be either malicious or compromised, can subsequently send messages to exploit a vulnerable extension. While a compromised web page is in itself troublesome, a compromised extension is much more powerful, as a Web attacker can leverage the extension's privileges to attack unrelated sites or exfiltrate sensitive user data to arbitrary sites. In the second scenario, an attacker can trick a user into installing a specific extension under their control. As previously, this extension would send the payload to exploit a vulnerable extension (i.e., the confused deputy). A malicious extension using this technique would be harder to detect than a classical malicious one: as it does not need any permission nor uses any sensitive API, its maliciousness stays hidden [6]. The only aim of such a malicious extension would be to exploit the privileges of vulnerable ones. As a cover, it could implement innocuous functionality that does not require any privilege, making it easy to pass through the review process [16]. To evaluate the feasibility of an attack through an unprivileged extension, we uploaded such an extension to the Chrome Web Store. Under the pretense of customizing the default new tab page in Chrome, our extension was sending malicious payloads to exploit two vulnerable extensions reported by DoubleX. Our extension was reviewed, and we were notified of its acceptance one day later. Once accepted, we installed the extension along with the two vulnerable ones. We confirm that we could exploit their vulnerabilities against ourselves. Similarly to CrossFire [6], we stress that our extension was designed as a case study. Specifically, we did not test it against real users, nor harm anyone; we set the extension visibility to *unlisted* (i.e., only people with the link could see it), we did not advertise it, and we confirm that it was downloaded only once (by us, to test it), and then we promptly removed it. Hence, we are confident that neither users were harmed nor details of the vulnerable extensions made public.

Naturally, for attackers to exploit a vulnerable extension, the victim should have the extension installed. An attacker can either detect the extensions installed by a given user (e.g., by leveraging DOM changes [75, 77], style changes [44], WARs [40, 70], or timing-channels [69]) to send tailored payloads or simply try to exploit a victim by firing all their malicious payloads. In this paper, we consider an extension to be vulnerable when at least another extension or web page can exploit its privileges to lead to the security or privacy issues we presented. This is motivated by the fact that any website could be compromised: even high-profile sites like Google had an XSS vulnerability (2019) [63].

4 DoubleX

DoubleX performs a fully static data flow analysis of browser extensions to detect those with suspicious external flows. We chose to conduct a static analysis due to its speed and code coverage. This section first provides a high-level overview of our system before presenting its three main components in more detail.

4.1 Conceptual Overview

As illustrated by Figure 3, DoubleX abstracts the source code of an extension with semantic information and models the interactions within and outside of an extension. This way, we can perform a data flow analysis to identify any path between an attacker and sensitive APIs. In its core, we implemented DoubleX in Python. First, we build an Abstract Syntax Tree (AST [2]) for each extension

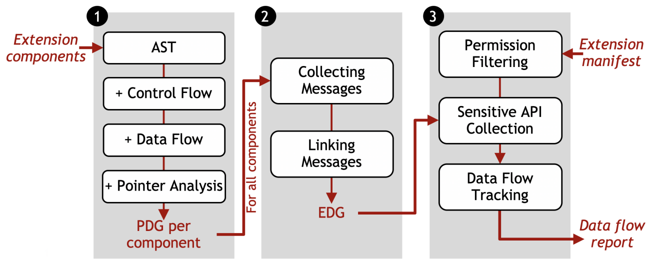

Figure 3: Architecture of DoubleX

component, which we enhance with control and data flows, and pointer analysis information. We refer to the resulting graph as a Program Dependence Graph (PDG) (Figure 3 stage 1). We adopt a definition of the PDG that slightly differs from Ferrante et al. [31], as we chose to add control and data flow edges to the AST. This way, we retain information regarding statement order and have a fine-grained representation of the data flows directly at variable level (Section 4.2). Next, we define a new graph structure, namely the Extension Dependence Graph (EDG), which models messages exchanged within and outside of an extension. (Figure 3 stage 2). For this purpose, we traverse each extension component's PDG and collect all messaging APIs. Based on the component and the API used, we can infer with whom it is exchanging messages. For internal messages, DoubleX links, e.g., the message sent by component *A* to the message received by *B* (for all components) with a *message flow*, to represent the interactions between the components. For external messages, DoubleX flags them as attacker controllable (Section 4.3). Overall, our EDG is a joint structure that includes control flows, data flows, and pointer analysis information at extension level and models interactions within an extension and with a web page/another extension (i.e., an attacker). Finally, we leverage our EDG to perform a data flow analysis targeting security- and privacy-critical APIs in extensions (Figure 3 stage 3). In particular, we consider integrity (attacker-controllable data enters a sink) and confidentiality (user sensitive data is exfiltrated) threats in tracking relevant data flows. DoubleX summarizes its findings in a fine-grained data flow report (Section 4.4).

4.2 Per-Component PDG Generation

To analyze a browser extension, we first abstract the code of each component independently. In particular, we model each component with a separate PDG, which includes AST edges, control and data flow edges, and pointer analysis information.

4.2.1 Syntactic Analysis First, DoubleX builds the AST of each component with Esprima [33]. We chose to rely on Esprima given its thorough set of test cases [34] and widespread use by prior work [6, 28, 29, 32, 45, 50, 65, 67, 73, 74, 80]. Esprima takes a valid JavaScript sample as input and produces an ordered tree (the AST) representing the program syntactic structure (i.e., the nesting of programming constructs). Next, to detect whether an extension executes attacker-controllable data or exfiltrates sensitive user information, we need a more complex abstraction of the code that goes beyond its syntactic order. Specifically, DoubleX gives more

```
1 b = 1;
2 if (b === 1) {
3   a = 2;
4 } else {
5   a = 3;
6 }
7 var c = a*a;
```

Listing 3: JavaScript code example

semantics to the AST nodes by (1) generating and storing their control flows, (2) their data flows, and (3) computing variable values.

4.2.2 Control Flow Analysis To reason about the conditions that should be met for a specific execution path to be taken, DoubleX extends the AST with control flows. To do so, we use the CFG (Control Flow Graph) implementation of Fass et al. [28, 29]. Flows of control are represented on statement nodes, which are connected with labeled and directed edges. Edges originating from predicates are labeled with a boolean, representing the value the predicate should evaluate to for its descendants in the graph to be executed. Furthermore, non-predicate statement nodes are connected with an ϵ edge. For example, Figure 4 (considering the blue dotted control flow edges) presents an execution path difference when the `if` condition is true vs. false. Still, the CFG does not enable us to infer whether the condition is true or not.

4.2.3 Data Flow Analysis To reason about variable dependencies and compute variable values, DoubleX adds data flow edges to the CFG, which becomes a PDG. In this paper, we did not use the data flow implementation from Fass et al., which did not fit our needs (e.g., no function argument passing nor pointer analysis). In the following, we describe our approach. Even though data flow and pointer analyses are interlinked and we perform them in the same CFG traversal, we present them in two sections, for clarity reasons.

To ease the value computation process, we represent data flows between `Identifier` nodes. In particular, we connect `Identifier` nodes (referencing, e.g., variables, functions, or objects) with a directed data flow edge if and only if they are defined or modified at the source node and used (or called) at the destination node, with respect to the scoping rules. If a variable is defined with the *window* object, directly assigned, or defined outside of any function, it is in the global scope. Otherwise, the variables can only be accessed in specific parts of the code (the local scope). To keep track of the variables currently defined and accessible in a given scope, DoubleX defines a list of *Scope* objects. In particular, we leverage CFG information to build different and independent *Scope* objects to handle variables from branches triggered by exclusive predicates (e.g., a *true* vs. *false* `if` branch) separately, to avoid impossible data flows. When exiting such a conditional node, we merge all variables defined or modified in the different branches to their corresponding scope (i.e., global or specific local scope) so that these variables are all known if further used. This way, DoubleX traverses the CFG and links the encountered variables to their definition or modification sites with a data flow edge. For example, the orange dashed data flow edges in Figure 4 represent variable dependencies. Specifically, we link variable *b* from its definition site (Listing 3 line 1) to its usage (line 2). The same applies to *a* (defined line 3 and used two times line 7). As our analysis is path sensitive for simple constraints (see Section 4.2.4), there is no data flow coming from *a*'s definition line 5.

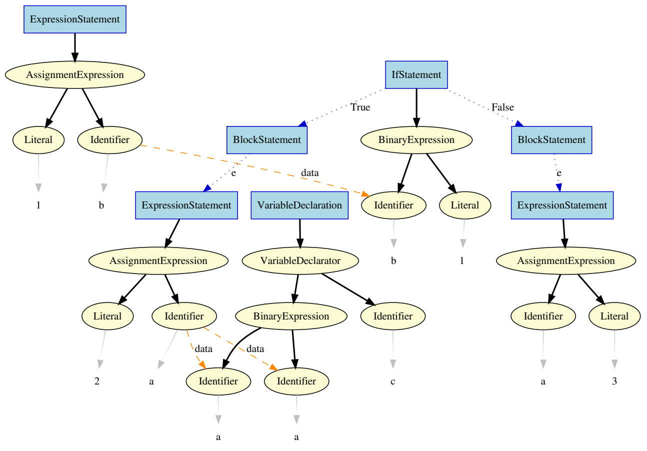

Figure 4: AST of Listing 3 extended with control & data flows

As far as functions are concerned, we hoist `FunctionDeclaration` nodes at the top of the current scope (as they may be first used then defined [25]) and distinguish them from `(Arrow)FunctionExpression` nodes (which have to be defined before usage) [56]. In addition, DoubleX respects function scoping rules, e.g., closures and lexical scoping. Also, we define a *parameter flow* to link function parameters at the call sites to their definition site, and we keep track of the returned values. This way, our analysis is inter-procedural, and we define our PDG at program level.

We chose to build the PDG by traversing the CFG one time (vs. iterating until we reach a fixed point), for performance reasons. While this may lead to under-approximations, our analysis stays accurate (low false-positive and low false-negative rates, cf. Sections 5.2.1 and 5.4) and is able to scale (cf. Section 5). We discuss drawbacks of our static approach in Section 6.2.

4.2.4 Pointer Analysis To determine variable values we follow four main principles: (1) if we already computed a node value, we fetch it from its `value` attribute, (2) we know the value of `Literal` nodes, which Esprima stores as a node attribute, (3) whenever there is a data flow between two `Identifier` nodes (from *source* to *destination*), the *destination* has the same value as the *source*, and (4) we define different rules to compute the values of variables, which undergo specific operations. Specifically, we handle assignment, arithmetic, string, comparison, and logical operators. We illustrate these principles in Algorithm 1, which is a simplified extract of our pointer analysis module. We call this script for specific nodes while we traverse the graph to perform the data flow analysis.

For example, in Figure 4, there is an `AssignmentExpression` (Algorithm 1 lines 11-17). Here, the `Identifier` *b* is declared. To compute its value, we determine the value of its symmetric node (line 15): as the `Literal` value is 1 (lines 3-4), so is *b* (line 16). Next, there is an `IfStatement`, whose condition is a `BinaryExpression` (lines 18-22). By following the data flow from *b* backward, we get its value 1 (lines 19 and 5-7). As the condition always evaluates to *true* (lines 20-22), DoubleX solely focuses on this branch (meaning that Listing 3 line 5 is never analyzed, hence no data flow from here, which limits over-approximations due to impossible cases). Finally, DoubleX

```
Data: node object $n$
Result: $n$ computed value
if $n.value$ is not None then
    return $n$.value;
else if $n.name$ == 'Literal' then
    $value \leftarrow n$.attributes['value'];
else if $n.name$ == 'Identifier' then
    if $n$ is the destination of a data flow from $source$ then
        $value \leftarrow source$.value;
    else
        $value \leftarrow n$.attributes['name'];
    end
else if $n.name$ in ('VariableDeclaration', 'AssignmentExpression') then
    $value \leftarrow$ None;
    find the defined/assigned Identifier nodes; // variable names
    for each Identifier node $i$ do
        $p \leftarrow$ calculate the symmetric path to $i$;
        // + some refinements for Array, Object nodes etc.
        $i$.value $\leftarrow$ call compute_value($p$)
    end
else if $n.name$ == 'BinaryExpression' then
    $operand1 \leftarrow$ compute_value($n$.children[0]);
    $operand2 \leftarrow$ compute_value($n$.children[1]);
    $operator \leftarrow n$.attributes['operator'];
    $value \leftarrow operand1\ operator\ operand2$;
else if ... then
    ...
$n$.value $\leftarrow value$;
return $value$
```

Algorithm 1: Pointer analysis extract: *compute_value*

fetches the value 2 from a's symmetric node and computes the operation to get the value 4 for c.

For clarity reasons, we chose a simplistic example. In particular, DoubleX also analyzes asymmetric variable declarations, e.g., in cases of arrays, objects, or functions, and destructuring assignments. Regarding objects and arrays, we store a handler to their definition site. This way, whenever a specific property is used or modified, we can follow the data flows to access the definition site, traverse the corresponding sub-AST to find the property/index, and compute its value. If the object or array is defined on the fly, we store its components in a dictionary, which becomes the handler to the object/array. Finally, whenever a function is called, we follow the data flows to find the function definition site. DoubleX then passes the function parameters at the call site(s) to the definition site by leveraging the parameter flows (cf. Section 4.2.3), before retraversing the function. For reproducibility and reviewing purposes, we make our source code available [27]. Overall, our pointer analysis enables us to add more semantic information to our data flow analysis. Specifically, it enables us to handle aliased variables and detect sensitive or messaging APIs not written in plaintext, as highlighted and discussed in Sections 4.3.1 and 4.4.2.

4.3 Generating the EDG

In the previous step, DoubleX generated the PDG of each extension component independently. Still, to detect suspicious data flows in an extension, we also need to understand the intricate relations between its components and detect external messages. To this end, we collect all messages sent and received by each component and order them per messaging API. This way, DoubleX leverages the APIs to know (for a given message) which components are communicating (or if the message is coming from a web page/another extension).

	Content script	Background page
message1	`sent`: *"Hi BP"*	`received`: *request*
message2	`got-response`: *response*	`responded`: *{farewell: "Bye CS"}*

Table 1: Message collection entry for the extension of Listing 2 (channel: one-time, deprecated APIs: no)

In the case of internal messages, we subsequently link the message sent by component A to the message received by component B with a *message flow*. Our EDG summarizes all this information.

4.3.1 Collecting Messages To collect the messages exchanged within and outside of an extension, we traverse the PDG of each component and look for specific messaging APIs. We consider all APIs presented in Section 2.2 (both for the Chromium-based browsers and Firefox), as well as deprecated APIs, which Chromium still supports (at least until Manifest V3 [19]), e.g., `chrome.extension.sendMessage` [11, 13]. Since we compute node values with DoubleX pointer analysis, we can also detect messaging APIs not written in plain text, e.g., string concatenation or referred to over aliases. Once we found a message-passing API, we look for the specific message that is sent (with a distinction between an initial message, `sent`, and a response, `responded`) or received (similarly, getting a message, `received`, or a response, `got-response`). For this purpose, we created an abstract representation of each API (based on the official documentation from Chrome and Mozilla) to know, depending on the number of arguments, which parameter corresponds to the message. For example, in Listing 2, the first parameter of `chrome.runtime.sendMessage` is the message sent by the content script, while the second one is a callback to receive the response from the background or WAR [12] (the API used also indicates which components are communicating). Once we know the message position, we collect the message. It can either be directly accessible (e.g., "Hi BP" Listing 2 line 2), or accessible only after callback resolution, which DoubleX performs by following the data and parameter flows (e.g., the callback `sendResponse` is defined Listing 2 line 8 and called line 10 with the message as first parameter). In addition, DoubleX analyzes `Promise`, such as calls to `.then` or `.resolve`, which Firefox can use instead of callbacks like Chromium. Finally, we store the collected messages. For internal messages, we order them per pair of components, one-time vs. long-lived channels, and deprecated API usage or not. Table 1 sums up the messages that DoubleX collected. As indicated in Listing 2, the content script sends one message and gets a response, while the background page gets a message and responds. For external messages, we keep track of which extension component received (or sent) a message, at which line in the code, and store the corresponding node object for future analyses (cf. Section 4.4.3). For example, in Listing 4, DoubleX reports the external message *event*, received line 1 in the content script.

4.3.2 Linking Messages Next, for internal messages, we link the messages sent (`sent`, `responded`) by a component to the messages received (`received`, `got-response`) by the other component, i.e., we connect a sender node to the corresponding receiver node. This way, DoubleX joins the individual component PDGs with a *message flow*. We give a graphical example in Figure 6 in the Appendix.

Besides, to keep track of variable values, we update the values of the receiver nodes (and those depending on it) with the sender node values. For the example of Table 1, we now know the following values: *request = "Hi BP"* and *response = {farewell: "Bye CS"}*. Thus, we can compute the value of *csReceived* (Listing 2 line 3) by leveraging the `ObjectExpression` *{farewell: "Bye CS"}* with the key *farewell*, getting "Bye CS" (cf. pointer analysis from Section 4.2.4). Similarly, the value of *bpReceived* (line 9) is *"Hi BP"*. Overall, DoubleX statically produces a graph structure, which gives an abstract semantic meaning to the extension code (including control and data flows, and pointer analysis) and models interactions within and outside of an extension. We refer to this graph as the EDG.

4.4 Detecting Suspicious Data Flows

Finally, DoubleX leverages the EDG to detect and analyze suspicious data flows. First, and for each extension, we prefilter sensitive APIs (i.e., that an attacker could exploit to gain access to an extension's privileged capabilities) based on an extension's permissions. Then, we traverse the EDG to collect any prefiltered sensitive APIs. As we flagged external messages in Section 4.3.1, we can finally perform a data flow analysis (from source to sink) to detect if these sensitive APIs are executed with attacker-controllable data or could exfiltrate sensitive user data.

4.4.1 Permission Filtering Given that an extension cannot be exploited without the corresponding permissions, DoubleX parses the manifests to automatically generate, for each extension, a list of sensitive APIs that the extension is allowed to access. We list the sensitive APIs we consider in Table 5 in the Appendix. In practice, for `XMLHttpRequest` and its derivatives (such as `fetch`), we only consider extensions that are allowed to make requests to arbitrary hosts (e.g., through the <all_urls> permission). For the exfiltration APIs, such as `bookmarks.getTree and history.search`, we ensure that the extensions are allowed to access them (i.e., *bookmarks* and *history* permission). Finally, for `tabs.executeScript`, we verify that the extensions either have *host* or *activeTab* permission [21], to execute code in the active tab. In addition, for the code execution APIs (e.g., `eval` or `tabs.executeScript`) in high-privilege components, we verify that the extensions define a Content Security Policy (CSP) [53] that allows their invocations. We also take into account some specificities of Chrome manifest V3 (such extensions are accepted in the Chrome Web Store since January 2021 [20]), e.g., *host* permissions are stored separately from other permissions or `XMLHttpRequest` is not defined in service workers [19].

4.4.2 Sensitive API Collection To perform its intended functionality, an extension may use specific APIs, which lead to security or privacy issues when attackers can exploit them. The extension component presenting the vulnerability will determine the attack surface. While background page and WARs are highly privileged, content scripts have less capabilities. For this reason, we consider different sensitive APIs, depending on the components we are analyzing (as specified in Table 5 in the Appendix). As explained in Section 4.4.1, we only consider APIs that have relevant permissions.

We detect these APIs by traversing the EDG and computing node values. Whenever DoubleX reports a sensitive API, we store the API name, the node object, and its corresponding value for further analyses (cf. following section). Besides, we keep track of the extension component, which uses the sensitive API and the corresponding line number. For example, DoubleX accurately reports the call to *eval* in Listing 4 line 2. Even though it is not written in plain text and is dynamically invoked, our pointer analysis module computes the correct value (i.e., string concatenation, recognizes the access of the `eval` function property of the *window* object with the bracket notation, recognizes the call to the sensitive API `eval`). We also report *eval* line 4. Both reports are stored as sensitive APIs for the content script.

4.4.3 Data Flow Tracking After collecting external messages, i.e., messages exchanged with an attacker (Section 4.3.1) and detecting relevant sensitive API invocations (Section 4.4.2), DoubleX performs a data flow analysis (from source to sink). Here, we aim at finding any path between dangerous or sensitive data and security- or privacy-critical APIs. Based on the way they operate, we distinguish three categories of sensitive APIs (which we refer to as *danger*):

• **Direct dangers** can directly leverage attacker-controllable data as parameter to perform malicious activities. Such APIs include `downloads.download` and `tabs.executeScript`, knowing that only the high-privilege components can call them (while `eval` can also be used by the content scripts). To handle such APIs, DoubleX extracts their parameters so that it can verify if they depend on data coming from an attacker. To limit false positives, we only extract the *relevant* parameters. For example, it is dangerous to have an attacker-controllable input in the second parameter of `tabs.executeScript` (or the first parameter if the tab ID is not indicated), provided it contains the code to be executed [60]. The first parameter, though, only allows to choose the tab to execute a script in, which is worthless for attackers if they cannot control the code.

• **Indirect dangers** work in two steps: first, they have to be called with attacker-controllable data; second, they need to send the results back to the attacker. For example, to perform cross-origin requests, all components can use `fetch` or `ajax`. We analyze these APIs in two steps: first, we verify if an attacker can control the relevant API parameters and if it is the case; second, we verify whether the data sent back to a web page or extension (i.e., the attacker) depends on data the extension received.

• **Exfiltration dangers** directly exfiltrate sensitive user data and do not necessarily need any input from an attacker. Such APIs can only be used by the high-privilege components and include `cookies.getAll`, `bookmarks.getTree`, and `history.search`. DoubleX extracts the sensitive API callback parameters and analyzes if they are sent back to an attacker.

As the EDG models interactions between extension components, we can handle cases where messages with attacker-controlled data (or data to be exfiltrated) are forwarded back and forth between the extension components before being exploited. Finally, DoubleX summarizes its findings in a data flow report. Specifically, it indicates the sensitive APIs it found per extension component (including API name, line number, and computed value) and if it detected a suspicious data flow. When it is the case, the report indicates if data was received from or sent to an attacker (including in which extension component, the line number, API value, etc.).

For example, Listing 5 is a simplified extract of the data flow report for Listing 4 (the full report is in Listing 7 in the Appendix). In particular, the *value* entry line 2 indicates that DoubleX accurately

```
1  addEventListener('message', function(event) {
2    window['e' + 'v' + '' + 'al'](event.data);
3    event = {'data': 42};
4    eval(event.data);
5  })
```

Listing 4: Vulnerable content script example

```
1  {"direct-danger1": "eval",
2  "value": "window.eval(event.data)",
3  "line": "2 - 2",
4  "dataflow": true,
5  "param1": {
6      "received": "event",
7      "line": "1 - 1"}},
8  {"direct-danger2": "eval",
9  "value": "eval(42)",
10 "line": "4 - 4",
11 "dataflow": false}
```

Listing 5: Extract of the data flow report for Listing 4

computes the first call to `eval`, despite string concatenation and dynamic invocation. Lines 2-7 show that it detects a data flow between the first parameter of `eval` Listing 4 line 2 and the message *event* received Listing 4 line 1, from a web page. Thus, we report here that the sensitive API `eval` can be called with attacker-controllable data. The second danger entry (lines 8-11) revolves around the second call to *eval(event.data)* (Listing 4 line 4). DoubleX detects that the *event* object has been redefined, as it computes the value 42 for *event.data* (line 9) and labels the danger as not having an externally controllable data flow (line 11). This way, the combination of our data flow and pointer analyses enables us to accurately label the first call to *eval(event.data)* as vulnerable without misclassifying the second one.

5 Large-Scale Analysis of Chrome Extensions

To evaluate the precision and recall of DoubleX regarding suspicious data flows detected, we apply it to Chrome extensions. In the following, we first outline how we collected 154,484 extensions and extracted their components. Subsequently, we describe our large-scale data flow analysis results with a focus on the vulnerable extensions we found. Furthermore, we compare DoubleX to directly related work and evaluate our approach on a labeled vulnerable extension set. Finally, we discuss DoubleX run-time performance and summarize our main findings.

5.1 Extension Collection and Setup

We designed DoubleX to analyze both Chromium-based and Firefox extensions. In this section, we focus solely on Chrome, while we discuss and analyze Firefox extensions in Section 6.3. We first report on our extension set before discussing the number of extensions we could analyze.

5.1.1 Collecting Extensions To collect extensions, we leveraged the Chrome Web Store sitemap [10], which contains links to all extensions. Out of the 195,265 listed extensions, we could successfully download 174,112 of them on March 16, 2021. The remaining ones were either not available for download for an OS X user agent, or only available for sale. Also, 19,628 downloaded extensions were themes, i.e., had no JavaScript component [22]. Thus, we retain 154,484 extensions for further consideration.

	#Analyzed	#Parsing errors	#Timeouts/Crashes
Extensions	154,484	3,674	9,500
- Content scripts	65,047	1,871	5,586
- Background page	98,974	1,847	4,227
- WARs	7,668	597	1,454

Table 2: Analyzed Chrome extensions

For each extension, we parsed its `manifest.json` [17] to extract the source code of its components. Even though DoubleX can analyze UI pages, they are not part of our threat model, as they cannot be forcefully opened (i.e., an attacker cannot deliver their messages to them). In the following, we consider content scripts, background page, and WARs. Specifically, we combined all content scripts into a single JavaScript file. For the background page, we considered both the content of the HTML background page and the scripts listed in the manifest background section. As for WARs, we collected all HTML files flagged as accessible and extracted both inline and external scripts. We chose to remove jQuery files (based on the output of `retire.js` [64] and file names such a *jquery-3.5.1.js*), to not analyze the well-known library to avoid running into timeouts. To limit potential false negatives, we consider that if a parameter of an unknown function is attacker controllable, so is the output of the function. In practice, we do not have any false negatives due to the absence of jQuery (cf. Section 5.4), and this approximation leads to less than 3.3% false positives (cf. Section 5.2.1), which we deem acceptable. Finally, to ease the manual verification of our data flow reports, we leveraged `js-beautify` [46] to produce a human-readable version of each extracted file.

5.1.2 Running DoubleX To analyze the considered Chrome extensions, we ran DoubleX with pairwise combined content scripts/ background page and content scripts/WARs. This allows us to reason about the capabilities of a Web attacker (which either requires communication through external messaging APIs or via the content scripts) as well as the ability of other extensions to send messages (since for background page and WARs, we also analyze external messages). For performance reasons, we set a timeout of 40 minutes to analyze two components of an extension (in particular, we set four 10-minute timeouts for specific steps of our analysis). We discuss DoubleX throughput into more details in Section 5.5.

As indicated in Table 2, we could analyze 91.5% of the extensions *completely*. For 2.4%, Esprima reported errors (mostly related to syntax errors in the code or usage of the unsupported spread syntax), while 6.1% ran into a timeout or the resulting EDG crashed the Python interpreter. Nevertheless, DoubleX could analyze such extensions *partially*. While the parsing errors are specific to some extensions, the timeouts concern independent components. For example, even if the PDG generation of an extension's content script timed out, we could analyze the background page independently for vulnerabilities. While DoubleX analyzed between 88.5 and 93.9% of the content scripts and background page *completely*, Esprima encountered more parsing errors for WARs (7.8%), which also timed out more often (19%). By checking the size of the WARs, we noticed that they are larger than the other components as numerous

Sensitive API	#Reports	#DF	#1-way DF	#Exploitable
Code Execution	113	102	-	63
- eval	38	34	-	30
- setInterval	1	1	-	0
- setTimeout	18	15	-	1
- tabs.executeScript	56	52	-	32
Triggering Downloads	21	21	-	21
Cross-Origin Requests	95	75	11	49
- ajax	6	6	0	5
- fetch	4	4	0	3
- get	4	4	0	3
- post	1	1	0	1
- XMLHttpRequest.open	80	60	11	37
Data Exfiltration	80	77	-	76
- bookmarks.getTree	31	29	-	29
- cookies.getAll	23	23	-	22
- history.search	23	22	-	22
- topSites.get	3	3	-	3
Sum	309	275	11	209

Table 3: DoubleX findings on Chrome extensions

HTML files are exposed due to extensions allowing *all* files to be web-accessible.

5.2 Analyzing DoubleX Reports

Overall, and out of our 154,484 extension set, DoubleX reported 278 extensions as suspicious, which sums up to 309 suspicious data flows. In this section, we report on the exploitability of these flows and present case studies. Subsequently, we discuss the evolution of vulnerable extensions between 2020 and 2021 and discuss vulnerability disclosure.

5.2.1 Suspicious Data Flows Our main findings regarding suspicious and exploitable data flows are summarized in Table 3. For each sensitive API (with a subtotal per flaw category), we first indicate the number of data flow reports generated by DoubleX (#Reports). Subsequently, we present the results of our manual analysis, regarding the number of reports with a data flow between an attacker and a sensitive API (#DF), the number of reports with a data flow between an attacker and a sensitive API but not back to the attacker (#1-way DF),[4] and, finally, the number of reports that an attacker could exploit based on our threat model (#Exploitable). In particular, to assess the exploitability of the reported flows, two experts with 5+ years of JavaScript experience first went through the reports and extensions. As mentioned in Section 4.4.3, DoubleX produces fine-grained data flow reports. In particular, they contain precise information about extension components, line numbers, and corresponding computed values, where a potentially dangerous data flow was detected. Thus, we could directly look for the code logic at precise line numbers to verify what happens in practice with data from/to an attacker. For the flows that were trivially exploitable (e.g., an attacker-controlled message directly flows into a sink) or clearly wrong, we flagged the reports accordingly. For more complicated cases (e.g., multiple data flows or sanitization functions), we installed the extension and built a payload to exploit the extension locally.

For the code execution APIs, the majority of the reports (102 / 113) contains an attacker-controllable flow to a sink, and 63 can be confirmed as vulnerable. Regarding download triggering, all of our 21 reports have a verified dangerous data flow and could be exploited to download arbitrary files. For cross-origin requests, we can exploit 49 / 95 flaws, even though 75 have a confirmed dangerous data flow (both from and back to an attacker). In such cases, the attacker could only control a part of the URL (hence the data flow), while we aim at making *arbitrary* requests. We also observe 11 reports where an attacker could make any request but did not receive the response (but a status code, for example). Finally, regarding data exfiltration, we can exploit almost all dangerous data flows reported (76 / 80).

Overall, out of 309 reports, 275 (89%) have a confirmed full data flow between a sensitive API and an attacker. Of those, 209 are exploitable under our threat model. Regarding the 66 remaining reports with a verified, yet unexploitable, dangerous data flow, we could not build an exploit mostly for one of the following reasons. For 24 cases, exploitation was prevented by a sanitization function (e.g., `JSON.stringify` or `escape`) or additional checks (e.g., checking that a payload is a number or checking that a payload matches a specific value). For 20 reports (cross-origin requests), only a part of the URL was attacker controllable, while we aim at making *arbitrary* requests. For 12 cases, only predefined functions could be called with an attacker-controllable parameter, e.g., *eval(predefined(attacker))*. Since 11 cross-origin requests have a data flow from an attacker to the sink but not back to the attacker, we only have 23 reports without dangerous data flow. In such cases, we observe two common limitations. For 10 cases, we over-approximate data flows, e.g., we handle data flows at object level and not at property level or propagate a suspicious function parameter flow to its returned value (as discussed in Section 5.1.1). Second, for 6 cases, there is a confusion regarding the sender of a message labeled as attacker-controllable, e.g., a service worker or a WebSocket object instead of a web page or another extension. While our static analysis is neither sound nor complete, in the spirit of soundiness [47], we chose a trade-off between accuracy and run-time performance. Also, our approach is more oriented toward detecting suspicious data flows and less toward proving the absence of vulnerability.

Specifically, we could detect 184 vulnerable extensions, totaling 209 vulnerable data flows, and impacting between 2.4 and 2.9 million users. Notably, almost 40% of these extensions can be exploited by *any* website or extension. Overall, 172 extensions are susceptible to a Web attacker, and 12 extensions are exploitable through an unprivileged extension. In addition, we could confirm 89% of the suspicious data flows reported by DoubleX, which highlights its high precision. Regarding vulnerable extensions DoubleX may have missed, we discuss DoubleX recall on a labeled vulnerable extension set in Section 5.4.

5.2.2 Case Studies Based on our data flow reports and findings, we now describe five case studies regarding vulnerable extensions that DoubleX detected. In doing so, we highlight the versatility of our tool in detecting non-obvious vulnerabilities.

[4]Only relevant for cross-origin request APIs, where an attacker aims at requesting arbitrary URLs and getting the response back from the server

Arbitrary Downloads with a Confused Deputy — The extension *eflehphffapiajamoknfnpfapdgaeffk* (10k+ users) registers an external message handler but does not specify the `externally_connectable` field in its manifest. Therefore, the handler accepts messages from any extension. The messages are then forwarded to several functions before ending in the *url* property of the `downloads.download` API, which allows an attacker to download arbitrary files. This example highlights the dangers of implicitly allowing any extension in externally connectable message handlers.

Arbitrary Code Execution — The extension *cdighkgkcaadmonmbocgpcnenffjjdfc* (4k+ users) can be exploited by *any* website to execute *arbitrary* code in the extension context. In fact, the content script, which can receive messages from any website, forwards all messages to the background page. In the background page, the messages subsequently flow into the *code* property of `tabs.executeScript`, without any sanitization. This example highlights the dangers of trusting input data which can be provided by an attacker.

Cross-Origin Requests from WARs — The extension *kohfgcgbkjodfcfkcackpagifgbcmimk* (200k+ users) registers an external message handler in its WARs to communicate with *naturalreaders.com* and its subdomains. By sending messages to the WARs, this website can make arbitrary requests and leak their content. While this may be an intended functionality of the extension, as discussed in Section 3, if this website gets compromised, an attacker could leverage the extension's elevated privileges to make arbitrary requests to *any* website and leak their content. This example highlights the dangers of authorizing a website to make *arbitrary* cross-origin requests instead of preferring the more secure CORS [54] alternative, which would limit the resources that can be accessed.

Most Visited Website Exfiltration — The content script of the extension *lklfbkdigihjaaeamncibechhgalldgl* (700k+ users) is injected into web pages from *msn.com* and its subdomains. Similarly to *kohfgcgbkjodfcfkcackpagifgbcmimk*, if this website gets compromised, an attacker could send a specific payload to the content script. This payload triggers the sending of a message to the background page, asking for information regarding a user's most visited websites. The background subsequently sends this information to the content script, which forwards it to the attacker. This example highlights the fact that an XSS in a website can be used by an attacker to leverage the extension's elevated privileges to access user sensitive information from other and unrelated sites.

Arbitrary Cross-Origin Requests via `eval` and `$.post` — The extension *ecnobkadlbkbcdmaidnhigklogkidlhf* (100+ users) was flagged by DoubleX for having a data flow to `eval` in its content script. Inspecting the extension, we find that there is no direct data flow from an attacker to the sink, but a data flow from an attacker to the URL of a `$.post` request, for which the response is then sent to `eval`. Since the extension has the host permission `*://*/*`, the call to `eval` can be abused to conduct arbitrary cross-origin requests. This way, an attacker can provide a link to their site to select the code to be executed, and in turn, can use that code to then conduct cross-origin requests against arbitrary other hosts, thereby allowing a cross-origin read of any resource of their choosing. Furthermore, since the content script is injected into *all* visited pages, an attacker can simply lure the victim to their site and send a postMessage to trigger the chain reaction.

5.2.3 *Comparison Between 2020 and 2021*

In this section, we discuss the evolution of vulnerable extensions between 2020 and 2021. Specifically, we focus on the life cycle of vulnerable extensions, i.e., whether vulnerable extensions from 2020 are still in the store in 2021 and, if so, whether they are still vulnerable. To perform these analyses, we crawled the Chrome Web Store also on June 19, 2020. Of the 166,513 extensions we could extract, 132,231 (79%) were still present in the store on March 16, 2021 (and 65,546 had not been updated in this nine-month time frame).

Similarly to Section 5.2.1, we ran DoubleX on our 2020 extension set. Specifically, our tool flags 279 extensions (0.17%) as having a suspicious data flow, which is similar to our results from 2021 (278 / 154,484 extensions). These 279 suspicious extensions expand to 317 suspicious data flows. As previously, we manually reviewed all reports from 2020, and we confirm that 286 (90%) have a verified dangerous data flow between an attacker and the sensitive APIs we consider. As already highlighted for our 2021 extension set, DoubleX has a very high precision regarding flagged data flows. Besides, we could exploit 219 of these flows, which leads to 193 vulnerable extensions. While we found 184 vulnerable extensions in 2021, the overall number of extensions in the Chrome Web Store slightly decreased in 2021, so that the proportion of vulnerable extensions did not change between 2020 and 2021 (0.12% of extensions). Still, 30 extensions that were vulnerable in 2020 are not in the store anymore, and, as of March 2021, only 3 have been fixed (by removing permissions or the vulnerable API call; we discuss disclosure in the following section). While there are 19 new vulnerable extensions, which were not in the store in 2020, 5 extensions existed before but turned vulnerable in 2021 (3 due to permission changes, 1 due to the addition of a vulnerable API call, and 1 due to allowing the communication with web pages directly in the background page).

Overall, we observe that 87% of the extensions that are vulnerable in 2021 were already vulnerable in 2020 (even though half of them were updated in between). Thus, we need a system like DoubleX to prevent vulnerable extensions from entering the store in the first place (we discuss integrating DoubleX in Chrome's vetting process in Section 6.1), especially as they tend to stay in the store. This is confirmed by the fact that the majority of developers we contacted did not take any action after our disclosure, as discussed in the next section.

5.2.4 *Disclosure to Extension Developers*

Finally, we disclosed our findings to the corresponding extension developers. Due to the impact of the flaws, we focussed on the extensions that can be exploited by *any* website or extension, leading to, e.g., universal XSS or sensitive user data exfiltration to *any* website.

We first reported our findings, including PoC exploits, regarding vulnerable extensions from 2020. In October 2020, we contacted 22 developers via emails, 4 over contact forms, and reported 9 issues directly to Google when we did not have any contact information. Similarly, in May 2021, we reported 13 additional vulnerable extensions (we contacted 9 developers via emails and reported 4 issues directly to Google). As of July 2021, of the 48 vulnerable extensions we reported, 45 are still in the store. Of those, 13 have been updated since our disclosure, but only 5 have been fixed (300k+

users, 50k+ users, 3k+ users, 2k+ users, and 35 users). For example, one extension (300k+ users) was updated to remove the `<all_urls>` permission to only allow the sites related to the extension, thus limiting the damage to third parties).

5.3 Comparative Analysis

The only related work that leverages a similar threat model and conducted a large-scale analysis on Chrome extensions is EmPoWeb from Somé (2019) [72]. Hence, we contrast our results against EmPoWeb's here and defer discussion of additional related work to Section 7. To analyze extensions' susceptibility to attacks through external messages, EmPoWeb is based on a lightweight call graph analysis. In practice, it yields an extremely high number of reports to manually vet: of the 66k Chrome extensions analyzed, it flagged 3.3k as suspicious, and only 5% were confirmed to be vulnerable (after Somé's huge manual effort).

In this section, we compare DoubleX's findings with EmPoWeb's on our 154,484 extension set. To this end, we ran the open-source version of EmPoWeb [71], which we slightly patched to match our attacker model, i.e., so that EmPoWeb considers the same sensitive APIs as DoubleX and with the same permissions (e.g., we consider *arbitrary* cross-origin requests whereas, in its original version, EmPoWeb considered even unauthorized access to a single cross-origin URL). Similarly, we excluded our WARs reports from this analysis (hence this section's results may slightly differ from Section 5.2.1), as EmPoWeb does not consider WARs.

As expected, EmPoWeb flags significantly more extensions as suspicious than DoubleX: it reports 2,665 extensions compared to our 268 (corresponding to 4,379 reported flaws vs. 299). Since EmPoWeb does not rest on a data flow analysis to generate reports, it mostly over-approximates the presence of an external message and of a sensitive API as a potential flaw. With DoubleX, though, we would flag such an extension only if we can find a data flow between an attacker and a sensitive API, hence yielding significantly fewer reports and significantly fewer false positives. Naturally, this is also possible that EmPoWeb flags vulnerable extensions that DoubleX does not detect. Still, given the extremely low true-positive rate of EmPoWeb (5%) and the fact that DoubleX detects almost 93% of the vulnerable extensions that were released with the EmPoWeb paper (cf. Section 5.4), we are confident that the majority of EmPoWeb's additional reports are false positives. In an Open World model like ours, to quantify potential false negatives, we would have to manually review hundreds of extensions to find a few vulnerable ones we may have missed, which would not be feasible.

In addition to being significantly more precise than EmPoWeb, DoubleX also detects vulnerabilities that EmPoWeb misses. Specifically, if we consider the 204 reports that we found vulnerable after manual review, 27 of them (13%) are not reported by EmPoWeb. It is especially prevalent for the `cookies.getAll` API, where 7 / 22 flaws are not detected and `tabs.executeScript` (9 / 32). While this may appear counter-intuitive (as EmPoWeb rather flags almost all extensions that contain an external message and a sensitive API), EmPoWeb relies on string matching and on a fixed list of possible ways to invoke specific APIs, so that it cannot always detect, e.g., aliases or dynamic sink invocations. For example, after aliasing *BPMessenger=chrome.runtime.sendMessage*, EmPoWeb does not detect *BPMessenger* as a message-passing API anymore. Contrary to EmPoWeb, DoubleX pointer analysis can detect and handle such aliasing cases as well as calls by reference, and APIs not written in plain text, which leads to a higher number of vulnerabilities found. In addition, our data flow analysis enables us to determine if sensitive or dangerous data is being exchanged with an attacker, which significantly limits our false positives.

5.4 Evaluation on a Labeled Dataset

To evaluate DoubleX false negatives, we consider the dataset of vulnerable extensions released by Somé with EmPoWeb. His paper [72] provides a list of extension IDs and corresponding vulnerabilities. Of the 171 Chrome extensions he reported as vulnerable in 2019, 82 still existed on March 16, 2021. We collected these extensions, and, after manual analysis, we confirm that 73 / 82 are still vulnerable. These 73 extensions total 163 previously reported vulnerabilities. As Somé considered some APIs that are not part of our attacker model (e.g., `storage`-related APIs[5]), we added them to our sensitive API list (only for this experiment).

DoubleX detects all vulnerabilities for 62 / 73 extensions, which corresponds to the accurate detection of 151 / 163 flaws (92.64%). For the twelve missing flaws, four are related to dynamic arrays, such as invocations of a function through *handlers[event.message]*, which we cannot statically resolve. Four other cases are data flow issues related to circular references in objects. For the last four cases, the handler function invokes a function that is not defined at this point in the parsing process. While DoubleX correctly hoists function declarations, this occurs when a function is defined as a variable (i.e., *foo = function() {...}*), which should be defined *before* use, according to the ECMAScript specification [25, 56]. In addition, for six extensions, which have not been updated since Somé's analysis, we report (and confirm, after manual review) three `XMLHttpRequest` and four `storage` vulnerabilities, which had not been found previously. This way, besides accurately detecting vulnerabilities in the wild (89% verified reported data flows, cf. Section 5.2.1), DoubleX also correctly flags the majority of known flaws (92.64%).

5.5 Run-Time Performance

Finally, we evaluated DoubleX run-time performance on a server with four Intel(R) Xeon(R) Platinum 8160 CPUs (each with 48 logical cores) and a total of 1.5 TB RAM. Since DoubleX runs the analysis of each extension on a single core, the run-time reported is for a single CPU only. The most time-consuming step of our approach is related to the data flow and pointer analyses. These operations naturally highly depend on the AST size, as we traverse it to store the variables newly declared or look for variables previously defined, and we re-traverse functions when they are called. On average, DoubleX needs 11 seconds to analyze an extension with content scripts and background page; and 96.5 seconds for content scripts and WARs (as the WARs are larger, cf. Section 5.1.2; Figure 7 in the Appendix presents DoubleX run-time performance depending on the extension size). Still, the corresponding median times are 2.5 and 31.8 seconds, while the maximum amount of time are 1,498 and 1,116 seconds. In practice, our average results are heavily biased by a few extensions, whose analysis lasted a long time. Figure 5 presents the Cumulative Distribution Function (CDF) [49] for our

[5]Storing and extracting data from an extension storage is not part of our attacker model, as we cannot assess to what extent this may cause damage

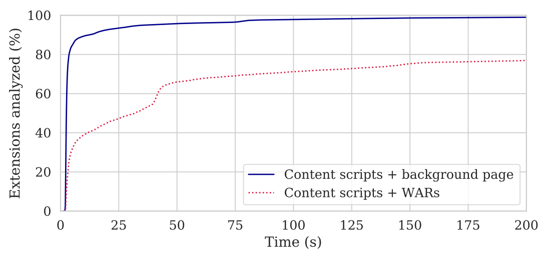

Figure 5: Run-time performance of DoubleX

run-time performance. In particular, we could analyze 93% of our extension set for content scripts and background page in less than 20 seconds and 45% for the content scripts and WARs. This way, DoubleX can effectively analyze extensions from the wild, with an analysis time of mostly a few seconds per extension.

5.6 Summary

To sum up, out of the 154,484 Chrome extensions DoubleX analyzed, it reported only 278 (0.18%) as having a data flow between an attacker and the sensitive APIs we considered. These suspicious flows expand to 309 reports, 275 (89%) of which have a verified dangerous data flow. Therefore, DoubleX is highly accurate to detect suspicious data flows. In addition, we verified that we could exploit 209 reports, according to our threat model. These 209 flaws correspond to 184 vulnerable extensions, with a total of over 2.4 million users. Regarding vulnerable extensions we may have missed, we evaluated DoubleX on the vulnerable extension set provided by EmPoWeb, where we accurately flag almost 93% of the flaws.

In addition, we observed that 87% of the vulnerable extensions were already in the store and vulnerable one year ago (despite disclosure and half of the extensions being updated in between). As extension developers do not necessarily have any incentive to patch vulnerable extensions, we believe that DoubleX could be integrated into the vetting process already conducted by Google (cf. Section 6.1). This would also delegate the responsibility of having vulnerable extensions in the store to Google. In addition to this, we envision that DoubleX could provide a feedback channel to developers, e.g., regarding the execution of attacker-controllable input or sensitive user data exfiltration, to limit having such vulnerabilities in the first place.

6 Discussion

In this section, we envision the option to integrate DoubleX into Chrome's vetting process. Then, we discuss limitations of our approach before considering its applicability to other ecosystems like Firefox extensions.

6.1 Extension Vetting: Workflow Integration

Given the precision and recall of DoubleX, we believe that it can be integrated into the vetting process already conducted by Google for newly uploaded extensions [16]. Currently, this system aims at identifying extensions that request powerful permissions or are clearly malicious, e.g., by spreading malicious software. Still, we envision that a feedback channel to alert developers regarding potential vulnerabilities would be relevant. It is particularly important for extensions that have privileges such as `<all_urls>`, which would allow an attacker who can exploit them to make arbitrary and authenticated requests and leak their content. As Google readily points out, high-privilege extensions require a more thorough analysis. Therefore, the information about vulnerabilities can also be of interest to the auditor, to limit the number of vulnerable extensions entering the store. This is all the more important as we noticed that very few developers acknowledged and fixed the vulnerabilities we reported (cf. Section 5.2.4).

6.2 Limitations

Regarding our extension set, we considered only the scripts which are part of the extension package, and we did not take import statements into account. Also, we chose not to analyze modules, as we are looking for vulnerabilities directly in the extension components.

As for our approach, DoubleX rests on a static analysis to build the EDG, including control, data, and message flows, and pointer analysis. While static analysis provides complete coverage of the available code, it is subject to the traditional flaws induced by JavaScript dynamic character [3, 30, 36, 37, 79]. For example, we may miss flaws due to dynamic code generation. While we can handle dynamically invoked sink functions (e.g., *window['e' + 'v' + '' + 'al'](value)*, cf. example in Section 4.4.2), as long as we can statically resolve and/or compute the arguments, we may miss dynamic function invocations (e.g., *handlers[partOfMessage])*, as discussed in Section 5.4. In addition, we chose to build the PDG of extension components by traversing the CFG one time (vs. iterating until we reach a fixed point; cf. Section 4.2.3), which may lead to under-approximations. We discussed the concept of soundiness in Section 5.2.1, though. Besides, as argued in Section 5.3, in an Open World model like ours, it would be extremely challenging to determine the number of vulnerable extensions DoubleX misses, as we would have to manually review hundreds of extensions to find a few vulnerable ones. As a best-effort strategy, we evaluated DoubleX on the vulnerable extension set released with EmPoWeb, where we accurately detect 92.64% of the flaws (naturally, if both DoubleX and EmPoWeb missed a vulnerable extension, there is no way for us to tell that). In addition, our tool is very precise, with over 89% of our reports which have a verified dangerous data flow.

6.3 Analyzing Firefox Extensions

Besides Chromium-based extensions, DoubleX can also analyze Firefox. To collect these extensions, we visited the Firefox gallery, which contains links to all extensions, ordered per category [51]. We used Puppeteer [68] to automatically download and unpack the extensions. We crawled the store on April 6, 2021, and could successfully collect 19,577 extensions. As for Chrome, we parsed the `manifest.json` of each extension to extract their components and ran DoubleX on them. Table 4 summarizes our findings. Out of 24 reports, we detected 8 that are exploitable under our threat model. In addition, we verified the presence of 22 dangerous data flows and 2 additional data flows without a backchannel. For example, and as previously, we consider that merely controlling a URL prefix for an `XMLHttpRequest` is not exploitable. As mentioned in Section 4.3.1, we took into account the specific message-passing APIs for Firefox and handled responses with a `Promise`. For the exfiltration APIs,

Sensitive API	#Reports	#DF	#1-way DF	#Exploitable
ajax	1	1	0	0
downloads.download	3	3	-	3
eval	2	2	-	0
fetch	4	3	1	1
setTimeout	5	5	-	0
tabs.executeScript	2	2	-	1
XMLHttpRequest.open	7	6	1	3
Sum	24	22	2	8

Table 4: DoubleX findings on Firefox extensions

though, we detect suspicious data flows based on callbacks and leave the `Promise` implementation for future work.

7 Related Work

In this section, we discuss prior work related to vulnerable extension analysis and data flow-based vulnerability detection.

Browser Extension Security — In 2010, Bandhakavi et al. introduced Vex, which leverages static information flow tracking on 2,452 (now deprecated) XPCOM [52] Firefox extensions [5]. With this XPCOM interface, though, they did not have the message-passing API problematic. In 2012, Carlini et al. combined a network traffic analysis of 100 Chrome extensions with a manual review to evaluate the effectiveness of Chrome security mechanisms [8]. In 2015, Calzavara et al. proposed a purely formal security analysis of browser extensions, looking for the privileges an attacker may escalate if a specific component was compromised [7]. In 2016, Salih et al. highlighted a security issue of the XPCOM namespace [6]. With CrossFire, they did a static data flow analysis to identify flows between globally accessible variables from extensions and security-sensitive XPCOM calls. In 2017, Starov et al. performed a dynamic analysis with BrowsingFog to detect privacy leakage from 10,000 Chrome extensions, showcasing that most leakage are not intentional [76]. Finally, in 2019, Somé considered message-passing APIs to exploit browser extension capabilities [72]. Still, his analysis, which merely relies on a lightweight call graph analysis and a fixed list of possible ways to invoke specific APIs, yields an extremely large number of false positives (of the 3.3k Chrome extensions he flagged, only 171 were vulnerable). In contrast, DoubleX defines an EDG to model the control, data, and message flows, including pointer analysis, inside an extension (as well as external messages). This graph then enables us to precisely reason about suspicious data flows, with regard to, e.g., aliasing, to detect non-obvious vulnerabilities.

While our approach targets *vulnerable* extensions, prior work also focussed on detecting *malicious* extensions, e.g., by monitoring their behavior [39, 81], detecting anomalous ratings [66], or tracking developer reputation [35]. Such malicious behaviors include stealing users' credentials, tracking users [82], spying on them [1], and voluntarily exfiltrating sensitive user information [9].

Data Flow Analysis for Vulnerability Detection — DoubleX can also be compared to systems using control and data flow tracking for vulnerability detection. For PHP, Jovanovic et al. implemented Pixy to perform a static data flow analysis to discover cross-site scripting vulnerabilities [38]. Yamaguchi et al. leveraged the AST enhanced with control and data flow information to model templates for known vulnerabilities with graph traversals and find similar flaws in other projects [83]. Backes et al. also used this data structure to identify vulnerabilities in PHP application [4]. Similarly, with VulSniper, Duan et al. leveraged control flow information to encode a program as a feature tensor and feed it to a neural network to detect vulnerabilities [24]. Contrary to these approaches, DoubleX does not need any information about previous vulnerabilities to operate.

8 Conclusion

In this paper, we designed and built DoubleX to detect security and privacy threats in benign-but-buggy extensions. In particular, we studied to what extent a web page or another extension without any specific privilege could exploit the capabilities of a vulnerable extension. To this end, DoubleX statically abstracts an extension source code to its EDG and performs a data flow analysis to detect suspicious flows between external actors (i.e., a web page or another extension, under the control of an attacker) and security- or privacy-critical APIs. The core components of DoubleX are the following. First, we abstract the source code of each extension component to its AST, which we enhance with control and data flows, and pointer analysis information. Second, we model messages exchanged between extension components with a *message flow*, and we collect messages exchanged outside of an extension (i.e., with an attacker). We refer to the resulting graph structure as the *Extension Dependence Graph (EDG)*. Finally, we leverage this graph to perform an in-depth data flow analysis between sensitive APIs in browser extensions and external messages.

We analyzed 154,484 Chrome extensions and flagged 278 as having a suspicious data flow. These suspicious flows expand to 309 reports, 89% of which have a verified dangerous data flow. This highlights the precision of our analysis. In addition, we detected 184 extensions that are exploitable under our threat model, leading to, e.g., arbitrary code execution in any website or sensitive user data exfiltration. Furthermore, we evaluated the recall of DoubleX on a ground-truth extension set, where it accurately flags almost 93% of known flaws. Finally, to raise awareness and enable developers and extension operators to automatically detect such threats before large-scale deployment, we make DoubleX publicly available [27].

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. In particular, we thank our shepherd Yousra Aafer for her guidance and support in preparing the final version of this paper.

References

[1] 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 *Euro S&P*.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (Second Edition).* Addison Wesley. ISBN: 978-0321486813.

[3] Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis for jQuery. In *Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA).*

[4] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *Euro S&P*.

[5] Sruthi Bandhakavi, Samuel T. Kingand P. Madhusudan, and Marianne Winslett. 2010. VEX: Vetting Browser Extensions for Security Vulnerabilities. In *USENIX Security Symposium.*

[6] Ahmet Salih Buyukkayhan, Kaan Onarlioglu, William Robertson, and Engin Kirda. 2016. CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities. In *NDSS.*

[7] 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*.

[8] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. 2012. An Evaluation of the Google Chrome Extension Security Architecture. In *USENIX Security Symposium.*

[9] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *CCS*.

[10] chrome. [n.d.]. Chrome Web Store Sitemap. https://chrome.google.com/webstore/sitemap. Accessed on 2021-04-25.

[11] chrome. [n.d.]. chrome.extension. https://developer.chrome.com/extensions/extension. Accessed on 2021-04-21.

[12] chrome. [n.d.]. chrome.runtime. https://developer.chrome.com/docs/extensions/reference/runtime. Accessed on 2021-04-29.

[13] chrome. [n.d.]. chrome.tabs. https://developer.chrome.com/extensions/tabs. Accessed on 2021-04-21.

[14] chrome. [n.d.]. Declare Permissions. https://developer.chrome.com/docs/extensions/mv3/declare_permissions. Accessed on 2021-04-21.

[15] chrome. [n.d.]. externally_connectable. https://developer.chrome.com/docs/extensions/mv3/manifest/externally_connectable/. Accessed on 2021-04-21.

[16] chrome. [n.d.]. How long will it take to review my item? https://developer.chrome.com/docs/webstore/faq/#faq-listing-108. Accessed on 2021-04-26.

[17] chrome. [n.d.]. Manifest File Format. https://developer.chrome.com/docs/extensions/mv3/manifest. Accessed on 2021-04-25.

[18] chrome. [n.d.]. Message Passing. https://developer.chrome.com/docs/extensions/mv3/messaging. Accessed on 2021-04-21.

[19] chrome. [n.d.]. Migrating to Manifest V3. https://developer.chrome.com/docs/extensions/mv3/intro/mv3-migration. Accessed on 2021-04-21.

[20] chrome. [n.d.]. Overview of Manifest V3. https://developer.chrome.com/docs/extensions/mv3/intro/mv3-overview/. Accessed on 2021-04-29.

[21] chrome. [n.d.]. The activeTab permission. https://developer.chrome.com/docs/extensions/mv2/manifest/activeTab/#what-activeTab-allows. Accessed on 2021-04-21.

[22] chrome. [n.d.]. Themes. https://developer.chrome.com/extensions/themes. Accessed on 2021-04-25.

[23] Chromium. [n.d.]. Changes to Cross-Origin Requests in Chrome Extension Content Scripts. https://www.chromium.org/Home/chromium-security/extension-content-script-fetches. Accessed on 2021-04-21.

[24] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. 2019. VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

[25] Ecma International. [n.d.]. ECMAScript 2020 Language Specification. https://262.ecma-international.org/11.0. Accessed on 2021-04-21.

[26] Extension Monitor. [n.d.]. Breaking Down the Chrome Web Store. https://extensionmonitor.com/blog/breaking-down-the-chrome-web-store-part-1. Accessed on 2021-04-25.

[27] Aurore Fass. [n.d.]. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. https://github.com/Aurore54F/DoubleX.

[28] Aurore Fass, Michael Backes, and Ben Stock. 2019. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *CCS*. Code repository: https://github.com/Aurore54F/HideNoSeek.

[29] Aurore Fass, Michael Backes, and Ben Stock. 2019. JStap: A Static Pre-Filter for Malicious JavaScript Detection. In *ACSAC*. Code repository: https://github.com/Aurore54F/JStap.

[30] Asger Feldthaus and Anders Møller. 2013. Semi-Automatic Rename Refactoring for JavaScript. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[31] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1987).

[32] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *NDSS*.

[33] Ariya Hidayat. [n.d.]. ECMAScript Parsing Infrastructure for Multipurpose Analysis. http://esprima.org. Accessed on 2021-04-29.

[34] Ariya Hidayat. [n.d.]. Esprima. https://github.com/jquery/esprima. Accessed on 2021-04-29.

[35] 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 *USENIX Security Symposium.*

[36] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the Eval That Men Do. In *International Symposium on Software Testing and Analysis (ISSTA)*.

[37] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *International Symposium on Static Analysis (SAS)*.

[38] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *S&P*.

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

[40] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. 2020. Carnus: Exploring the Privacy Threats of Browser Extension Fingerprinting. In *NDSS.*

[41] Jamila Kaya and Jacob Rickerd. [n.d.]. Security Researchers Partner With Chrome To Take Down Browser Extension Fraud Network Affecting Millions of Users. https://duo.com/labs/research/crxcavator-malvertising-2020. Accessed on 2021-04-27.

[42] Ravie Lakshmanan. [n.d.]. 49 New Google Chrome Extensions Caught Hijacking Cryptocurrency Wallets. https://thehackernews.com/2020/04/chrome-cryptocurrency-extensions.html. Accessed on 2021-04-27.

[43] Ravie Lakshmanan. [n.d.]. Over a Dozen Chrome Extensions Caught Hijacking Google Search Results for Millions. https://thehackernews.com/2021/02/over-dozen-chrome-extensions-caught.html. Accessed on 2021-04-27.

[44] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. 2021. Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets. In *USENIX Security Symposium*.

[45] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *USENIX Security Symposium.*

[46] Einar Lielmanis. [n.d.]. js-beautify. https://www.npmjs.com/package/js-beautify. Accessed on 2021-04-25.

[47] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. In *Communications of the ACM.*

[48] Joseph Menn. [n.d.]. Exclusive: Massive spying on users of Google's Chrome shows new security weakness. https://www.reuters.com/article/us-alphabet-google-chrome-exclusive/exclusive-massive-spying-on-users-of-googles-chrome-shows-new-security-weakness-idUSKBN23P0JO?il=0. Accessed on 2021-04-27.

[49] Katherine L. Monti. 1995. Folded Empirical Distribution Function Curves (Mountain Plots). In *The American Statistician.*

[50] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In *Dependable Systems and Networks (DSN)*.

[51] Mozilla. [n.d.]. Firefox Browser Add-ons: Extensions. https://addons.mozilla.org/en-US/firefox/extensions. Accessed on 2021-04-25.

[52] Mozilla Developer Network. [n.d.]. XPCOM Interfaces. https://developer.mozilla.org/en-US/docs/Archive/Mozilla/XUL/Tutorial/XPCOM_Interfaces. Accessed on 2021-04-29.

[53] Mozilla Developer Network. [n.d.]. Content Security Policy (CSP). https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/content_security_policy. Accessed on 2021-04-21.

[54] Mozilla Developer Network. [n.d.]. Cross-Origin Resource Sharing (CORS). https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. Accessed on 2021-07-30.

[55] Mozilla Developer Network. [n.d.]. EventTarget.addEventListener(). https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener. Accessed on 2021-04-21.

[56] Mozilla Developer Network. [n.d.]. Functions. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions. Accessed on 2021-04-29.

[57] Mozilla Developer Network. [n.d.]. manifest.json: permissions. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/permissions. Accessed on 2021-04-21.

[58] Mozilla Developer Network. [n.d.]. Promise. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. Accessed on 2021-04-21.

[59] Mozilla Developer Network. [n.d.]. Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Accessed on 2021-04-29.

[60] Mozilla Developer Network. [n.d.]. tabs.executeScript(). https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/tabs/executeScript. Accessed on 2021-04-21.

[61] Mozilla Developer Network. [n.d.]. WindowEventHandlers.onmessage. https://developer.mozilla.org/en-US/docs/Web/API/WindowEventHandlers/onmessage. Accessed on 2021-04-21.

[62] Mozilla Developer Network. [n.d.]. Window.postMessage(). https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage. Accessed on 2021-04-21.

[63] Tomasz Andrzej Nidecki. [n.d.]. Mutation XSS in Google Search. https://www.acunetix.com/blog/web-security-zone/mutation-xss-in-google-search. Accessed

on 2021-04-21.
[64] Erlend Oftedal. [n.d.]. Retire.js: What you require you must also retire. https://retirejs.github.io/retire.js. Accessed on 2021-04-25.
[65] 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*.
[66] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. 2020. You've Changed: Detecting Malicious Browser Extensions through their Update Deltas. In *CCS*.
[67] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. 2015. DexterJS: Robust Testing Platform for DOM-based XSS Vulnerabilities. In *Foundations of Software Engineering*.
[68] puppeteer. [n.d.]. puppeteer. https://github.com/puppeteer/puppeteer. Accessed on 2021-04-25.
[69] Iskander Sánchez-Rola, Igor Santos, and Davide Balzarotti. 2017. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In *USENIX Security Symposium*.
[70] Alexander Sjösten, Steven Acker, and Andrei Sabelfeld. 2017. Discovering Browser Extensions via Web Accessible Resources. In *Conference on Data and Application Security and Privacy (CODASPY)*.
[71] Dolière Somé. [n.d.]. extsanalyzer (EmPoWeb). https://gitlab.com/doliere/extsanalyzer. Accessed on 2021-04-29.
[72] Dolière Francis Somé. 2019. EmPoWeb: Empowering Web Applications with Browser Extensions. In *S&P*.
[73] Pratik Soni, Enrico Budianto, and Prateek Saxena. 2015. The Sicilian Defense: Signature-Based Whitelisting of Web JavaScript. In *CCS*.
[74] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. In *NDSS*.
[75] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. 2019. Unnecessarily Identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *WWW*.
[76] Oleksii Starov and Nick Nikiforakis. 2017. Extended Tracking Powers: Measuring the Privacy Diffusion Enabled by Browser Extensions. In *WWW*.
[77] Oleksii Starov and Nick Nikiforakis. 2017. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *S&P*.
[78] StatCounter. [n.d.]. Desktop Browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share/desktop/worldwide. Accessed on 2021-04-25.
[79] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. 2019. Static Analysis with Demand-Driven Value Refinement. In *ACM on Programming Languages*.
[80] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. 2017. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security. In *USENIX Security Symposium*.
[81] 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 *International Conference on Information Security (ISC)*.
[82] 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*.
[83] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *S&P*.

A Appendix

This appendix contains some supplementary material.

In particular, Listing 6 illustrates Chrome long-lived connections, as discussed in Section 2.2.

Listing 7 shows the full data flow report for the vulnerable content script example of Listing 4, as discussed in Section 4.4.

Table 5 lists the sensitive APIs that we considered for our large-scale analysis of extensions (provided an extension has the corresponding permissions, cf. Section 4.4.1). We indicate, in particular, if all extension components can access the sensitive APIs or only the high-privilege ones.

Figure 6 is a graphical representation of the EDG of the extension from Listing 2. As discussed in Sections 4.2 and 4.3, it includes control flows (blue dotted edges), data flows (orange dashed edges), and message flows (green solid edges).

Finally, Figure 7 presents DoubleX run-time performance depending on the extension size.

```
// Web page code
var port = chrome.runtime.connect({name: "myport"});
port.postMessage({greeting: "Hi BP"});

// Background page code
chrome.runtime.onConnectExternal.addListener(function(p) {
 p.onMessage.addListener(function(message) {
  received = message.greeting // Hi BP
 });
});
```

Listing 6: Messages: web page - background page (long-lived)

```
{
  "extension": "vuln-extension",
  "cs": {
    "direct_dangers": {
      "danger1": {
        "danger": "eval",
        "value": "window.eval(event.data)",
        "sink-param1": "event.data",
        "line": "2 - 2",
        "filename": "vuln-extension/content-script.js",
        "dataflow": true,
        "param_id0": {
          "received_from_wa_1": {
            "wa": "event",
            "line": "1 - 1",
            "filename": "vuln-extension/content-script.js",
            "where": "event",
          }
        }
      },
      "danger2": {
        "danger": "eval",
        "value": "eval(42)",
        "sink-param1": 42,
        "line": "4 - 4",
        "filename": "vuln-extension/content-script.js",
        "dataflow": false,
        "param_id0": {}
      }
     },
    "indirect_dangers": {},
    "exfiltration_dangers": {}
  },
  "bp": {
    "direct_dangers": {},
    "indirect_dangers": {},
    "exfiltration_dangers": {}
  }
}
```

Listing 7: Full data flow report for Listing 4

Flaw category	All components	High-privilege components
Code Execution	`eval, setInterval, setTimeout`	`tabs.executeScript`
Triggering Downloads		`downloads.download`
Cross-Origin Requests	`$.ajax, jQuery.ajax, fetch, $.get, jQuery.get, $http.get, $.post, $http.post, XMLHttpRequest().open, jQuery.post, XMLHttpRequest.open`	
Data Exfiltration		`bookmarks.getTree, cookies.getAll, history.search, topSites.get`

Table 5: Security- and privacy-critical APIs considered depending on the extension components

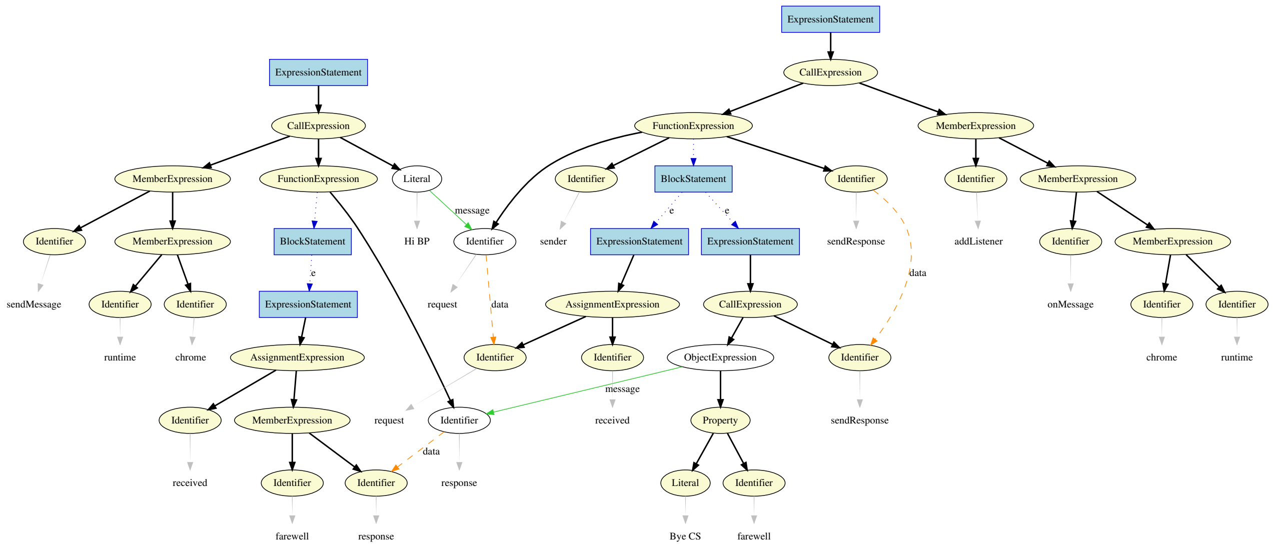

Figure 6: EDG of the extension from Listing 2 (see Table 1 for the specific messages exchanged)

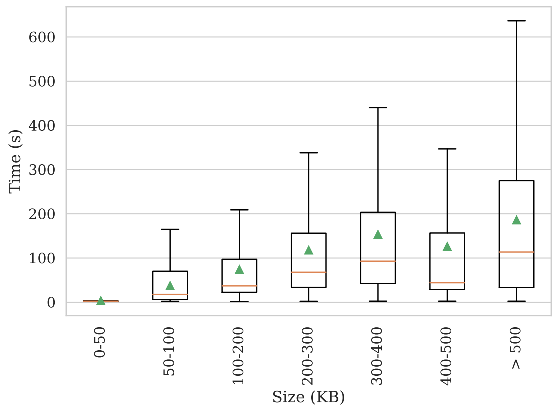

Figure 7: Run-time performance of DoubleX depending on the extension size (for content scripts and background page)