Mystique: Uncovering Information Leakage from Browser Extensions

Quan Chen
North Carolina State University
qchen10@ncsu.edu

Alexandros Kapravelos
North Carolina State University
akaprav@ncsu.edu

ABSTRACT

Browser extensions are small JavaScript, CSS and HTML programs that run inside the browser with special privileges. These programs, often written by third parties, operate on the pages that the browser is visiting, giving the user a programmatic way to configure the browser. The privacy implications that arise by allowing privileged third-party code to execute inside the users' browser are not well understood.

In this paper, we develop a taint analysis framework for browser extensions and use it to perform a large scale study of extensions in regard to their privacy practices. We first present a hybrid approach to traditional taint analysis: by leveraging the fact that extension source code is available to the runtime JavaScript engine, we implement as well as enhance traditional taint analysis using information gathered from static data flow and control-flow analysis of the JavaScript source code. Based on this, we further modify the Chromium browser to support taint tracking for extensions. We analyzed 178,893 extensions crawled from the Chrome Web Store between September 2016 and March 2018, as well as a separate set of all available extensions (2,790 in total) for the Opera browser at the time of analysis. From these, our analysis flagged 3,868 (2.13%) extensions as potentially leaking privacy-sensitive information. The top 10 most popular Chrome extensions that we confirmed to be leaking privacy-sensitive information have more than 60 million users combined. We ran the analysis on a local Kubernetes cluster and were able to finish within a month, demonstrating the feasibility of our approach for large-scale analysis of browser extensions. At the same time, our results emphasize the threat browser extensions pose to user privacy, and the need for countermeasures to safeguard against misbehaving extensions that abuse their privileges.

CCS CONCEPTS

• **Security and privacy → Browser security**; *Information flow control*;

KEYWORDS

Privacy, Browser Extensions, JavaScript, Taint Analysis, Information Flow

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243823

ACM Reference Format:
Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage, from Browser Extensions. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243823

1 INTRODUCTION

All popular web browsers today offer extension mechanisms that allow users to customize or enrich their web browsing experiences by modifying the browser's behavior, enhancing its functionalities or integrating with popular web services. To support interaction with the visited web pages, such as modifying their the contents or UI layouts, extension frameworks provide mechanisms to inject custom JavaScript code into a web page and execute in the page's context (e.g., [6, 39]). This capability allows extensions to inject code that retrieves private information from a web page (e.g., page URL, cookies, form inputs, etc). Moreover, browser extensions have access to privileged extension APIs that are out of reach from the normal JavaScript code executing as part of the web pages. For example, Chrome extensions can use the JavaScript extension API `chrome.history` to directly query any past browsing history information [9].

This unique vantage point enjoyed by browser extensions provide them opportunities to gain intimate knowledge of the browsing habits of their users; when this knowledge is abused, it puts users' privacy and personal information at risk. Although the potential for abuse is high, the privacy implications posed by browser extensions have only recently caught the attention of the security community. Several reports and blog posts shed light on the scope of the issue by manually analyzing a few extensions [24, 31, 51]. Recent works [43] and [52] investigated the privacy practices of browser extensions by analyzing the network traffic generated by extensions. Specifically, [43] applied heuristics to attempt decoding of common encoding/obfuscation techniques, while [52] used machine learning to identify traffic patterns that indicate possible privacy leaks. However, these previous efforts lack either the scale or the depth to examine the full scope of the privacy implications introduced by third-party extensions. For example, the approach proposed by [43] cannot handle customized encoding algorithms or encryption; traffic pattern analysis employed by [52] is prone to evasion whereby attackers mask their network traffic with noise. Indeed, addressing the potential privacy abuse posed by browser extensions requires not only an automatic analysis framework, but also a mechanism that tracks the detailed data flows inside browser extensions.

Requirements: Privacy-intrusive extensions abuse their privileges to leak sensitive information. To avoid detection at the network

level, they can arbitrarily generate decoy traffic patterns or otherwise obfuscate/encrypt such information before it is sent on the wire. Thus, to be generic an analysis framework must be able to label any sensitive information accessed by third-party extensions, as well as to track their usage throughout the lifetime of the extensions. That is, it must implement dynamic taint tracking (e.g., [27]). Such an analysis framework must be able to track data flows across all available JavaScript object types, handle control-flow dependencies, and address any browser-specific data flows paths that are introduced, for example, by the DOM interface or the extension APIs. Additionally, to detect extensions that utilize local storage to persist privacy-sensitive information for later exfiltration only when certain conditions are met (e.g., a threshold number of events), the analysis framework must also identify such extensions and flag them for further scrutiny.

Previous research in the direction of applying dynamic taint tracking to the browser context include [25, 26, 47], which relied on instrumenting the bytecode instructions emitted by the Firefox browser's JavaScript engine SpiderMonkey [40]. However, similar research efforts is lacking for the Google Chrome browser, which currently holds over 57.64% worldwide market share as of April 2018 [44]. Previous works [36, 38] implemented taint tracking for Chrome, albeit only for the string type and did not handle the extension API. There is currently no *complete* dynamic taint tracking implementation for Chrome browser's V8 JavaScript engine [29] that satisfies all the requirements of detecting privacy-intrusive extensions. Furthermore, given the highly optimized nature of the V8 engine, previous approaches that applied to Firefox cannot be straightforwardly adapted to Chrome.

Introducing Mystique: To help bridge this gap, we propose Mystique, an extension analysis framework that serves as the first effort at a complete implementation of dynamic taint tracking for the Google Chrome browser. We augment multiple components of the browser, particularly its V8 JavaScript engine, with taint tracking capabilities. To analyze extensions, Mystique automatically loads them in a monitored environment. Our primary goal in this paper is to identify third-party browser extensions that leak privacy-sensitive information. To this end, Mystique automatically taints values obtained from extension-accessible data sources that divulge users' private information. To overcome complexities of the JavaScript language as well as the V8 engine, Mystique implements runtime taint propagation by leveraging information obtained from a static data flow and control-flow dependency analysis of the JavaScript source code. Mystique logs extensions that triggered taint sinks with tainted values during the analysis. To aid in post-analysis understanding, Mystique also logs how tainted values are propagated and used by the extension JavaScript code.

We applied Mystique to analyze 178,893 extensions that were crawled from the Chrome Web Store between September 2016 and March 2018, plus a separate set of all available extensions (2,790 in total) for the Opera browser at the time of analysis. Our analysis flagged 3,868 (2.13%) extensions as potentially leaking privacy-sensitive information. The top 10 most popular Chrome extensions that we confirmed to be leaking privacy-sensitive information have more than 60 million users combined, highlighting the privacy threat posed by third-party browser extensions. From the analysis results, we also uncovered multiple encoding/obfuscation techniques employed by extensions. Our analysis of all 181,683 extensions was run on a local Kubernetes cluster and finished within a month, showing the feasibility of applying Mystique to large-scale analysis of browser extensions. Thus, Mystique can provide "analysis as a service", e.g., integrated as part of a triage system for online extension repositories such as the Chrome Web Store.

We release as open source software our dynamic taint tracking enhancements to the Chromium browser, as well as the framework to reproduce our experiments in this paper. We also provide a web interface that we have been using internally through which users can submit extensions to Mystique and get back the analysis results. More details can be found at https://mystique.csc.ncsu.edu/.

Contributions: The major contributions of this paper are:

- We propose a novel taint analysis technique that leverages both dynamic taint tracking and static analysis. We provide the first full implementation of hybrid taint tracking for the V8 JavaScript engine, based on techniques that leverage information gathered from static data flow and control-flow dependency analysis.
- We present Mystique, an analysis framework that builds on our hybrid taint tracking implementation to analyze and detect third-party browser extensions that abuse privacy-sensitive information.
- We conducted the first large-scale study of 181,683 third-party browser extensions in regard to their privacy practices.
- We advance the state of the art by uncovering obfuscation schemes used by extensions that escaped the attention of similar previous research efforts.

2 BACKGROUND

In this section, we first give an overview of the Chrome browser's extension framework, and the opportunities that this framework presents for extension authors to obtain and exfiltrate users' privacy-sensitive information. We also provide the relevant technical background of the V8 JavaScript engine that will be used for this work. Note that since Chromium is the open-source version of the Google Chrome browser, in the rest of this paper the names Chrome and Chromium will be used interchangeably.

2.1 Chrome Extension Framework

Chrome supports extensions that modify or enhance the functionality of the browser [14]. Extensions are written using JavaScript, HTML and/or CSS, and packaged together with a mandatory manifest file and distributed as a single zip archive. The manifest file describes the extension, and contains, among other parameters, the declared permissions that determine which Chrome extension API calls it can access, as well as what web pages (i.e., URLs) the extension can operate on. API permissions are typically requested by indicating the API names, while the allowed web pages is known as *host permissions* and are specified using match patterns. Match patterns allow wild-carding, and the special token `<all_urls>` matches any URL. We discuss the relevant aspects of the extensions architecture in the rest of this section. Please refer to Barth et.al [16] for a detailed treatment of the Chrome extension framework.

2.1.1 Content Scripts. Extensions can inject and run JavaScript code inside web pages. This injected code is known as content

script [6]. There are two ways to inject content scripts: 1) statically by declaring in the manifest the JavaScript files and their corresponding match patterns, so that Chrome automatically injects the scripts contained in the specified files into every web page whose URL satisfies one of the match patterns, or 2) dynamically by using the so-called programmatic injection method. Either way, the intent to use content scripts need to be declared (explicitly or implicitly) in the manifest file. For the former method, it is obvious that the JavaScript files containing the content scripts must be listed in the manifest file; for the latter method, the extension must: a) declare the "`tabs`" permission (in order to use the `chrome.tabs.executeScript` API), and b) also declare host permissions for the URLs where this injection should be allowed.

Since content scripts are injected into web pages and executed there, they run in the same environment as the normal JavaScript code that is downloaded as part of the web pages. For example, they have the same access to the Document Object Model (DOM) interface, and are therefore able to query it for page details or to make changes. Although content scripts are sandboxed from the normal JavaScript code of the web pages [6], they nevertheless provide a powerful feature and allow extensions the opportunity to access information that would otherwise not be available to them.

2.1.2 Background Pages. In addition to content scripts, extensions can also run JavaScript code in the background page [1], which is a special HTML page that is not visible to the user. Unlike content scripts, there can only be one background page per extension, and its purpose is to allow a long-running script to manage states for the lifetime of the extension. Background pages have full access to the Chrome extension API, as long as the appropriate permissions have been declared. For example, they can register callback functions to the `chrome.tabs.onUpdated` event, so as to be notified about the details of any tab update event (e.g., URL of a newly loaded page), provided they have declared the "`tabs`" permission and that the loaded page URL matches one of the host permissions.

2.1.3 Message Passing API. Given that content scripts execute inside web pages, there needs to be a way for them to communicate with the rest of the extension. This is achieved by using the message passing API [11], which allows callback functions to be registered that listen for messages being sent on the same channel. A message may be delivered to multiple registered callbacks; inside each of the callback functions further logic can be implemented to decide if actions should be performed for the received message. Both content scripts and background pages can register callbacks, as well as to send messages. The message passing API can also be used to communicate across extensions.

2.2 V8 JavaScript Engine

The Chromium browser uses V8 [29] to JIT-compile and execute JavaScript code. One characteristic of the V8 architecture is its use of two separate compilers, i.e., the full compiler (Full-codegen) and the optimizing compiler (Crankshaft). Both of them compile JavaScript to native code. Recently, V8 has moved away from this architecture in favor of an interpreter-compiler pair (i.e., Ignition and TurboFan) [46]. Nevertheless, the basic idea is still the same: begin executing JavaScript code with as little delay as possible, and during execution collect runtime information which will aid the optimizing compiler (Crankshaft or TurboFan) in generating efficient code for portions of the JavaScript source that are frequently executed. Our prototype implementation of Mystique (described in Section 4) is based on an earlier version of V8 that uses the Full-codegen/Crankshaft architecture, and we turned off the optimizing compiler (i.e., Crankshaft) so that JavaScript compilation is handled exclusively by Full-codegen. This implementation choice is primarily motivated by simplicity in a proof-of-concept prototype, as well as the intended "analysis as a service" usage scenario of Mystique. Nevertheless, we note that the JavaScript parser (and therefore V8's internal AST representation of JavaScript source code) is shared between Full-codegen and Crankshaft, so the methodology we present in this paper can also be adapted to Crankshaft. In addition, the code base for the JavaScript parser remains largely stable across V8 versions, and as a result our methodology can also be ported to the latest Ignition/TurboFan architecture.

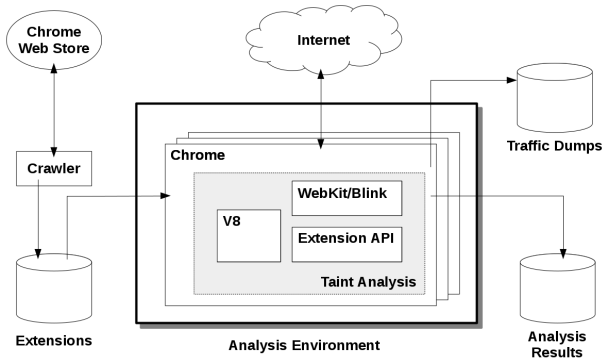

Figure 1: Architectural overview of Mystique, showing major components of Chrome that Mystique augments with taint tracking capabilities.

3 TECHNICAL APPROACH

Mystique utilizes dynamic taint analysis to track the runtime data flows inside third-party extensions and identify the ones that leak privacy-sensitive information. Specifically, it extends the Chromium browser and its JavaScript engine (V8) with taint tracking capabilities so that any values that can potentially contain privacy-sensitive information are marked (as tainted). To analyze an extension, Mystique launches an instance of our taint-enhanced Chromium browser with the extension preloaded, inside a monitored environment. The browser is then automatically driven to browse the web. Mystique logs any extension that triggers a taint sink with tainted values. Figure 1 shows the architectural overview of Mystique.

Previous research [36, 38] that implemented taint analysis for Chromium handled only string-to-string propagation. To detect privacy-leaking extensions, Mystique's taint analysis needs to be generic and therefore should consider all object types and data flow paths available to extension JavaScript code. To achieve this, Mystique needs to overcome the challenges that 1) JavaScript is a complex dynamically-typed language and as a result operations have different meanings that depend on object types; 2) the V8

JavaScript engine is highly optimized so it requires significant engineering efforts to patch all possible data flow paths such that they are taint-aware: for example, V8 can emit native code differently for arithmetic operations (e.g., "+") as either integer or floating-point operations depending on the operand values, despite JavaScript having only one (floating) number type; and furthermore 3) additions to the JavaScript semantics made by both the Chrome extension API [9] and the DOM interface create data flow paths that are not reflected at the JavaScript source level.

To address them, Mystique does not attempt to instrument each individual data-flow operation in order to propagate taint, as was done by Lekies et al. [36] and Melicher et al. [38] for Chromium (and also [25, 26, 47], which were for Firefox). This design choice primarily follows from the first two challenges, which together imply that it is not feasible to manually identify and patch all the possible data flow operations that can be emitted by the V8 JavaScript engine. On the other hand, since JavaScript is an interpreted language, all the JavaScript source code that is to be executed will become available to the runtime interpreter or JIT compiler at some point. This provides Mystique the opportunity to combine dynamic taint tracking with static source code analysis. Specifically, Mystique leverages information from static data flow and control-flow dependency analysis at the JavaScript source level to determine which additional objects should also be tainted, given a set of already-tainted objects. For simplicity, Mystique currently employs a flow-insensitive, intra-procedural analysis for this purpose. In the rest of this section, we present the details of Mystique's taint analysis framework.

3.1 Sensitive Data Sources

There are two broad categories of sources from which an extension can obtain users' privacy-sensitive information: 1) the DOM interface, and 2) the Chrome extension API. The DOM interface of a web page is accessible to all JavaScript code executing in it, including the content scripts injected by extensions. For example, the DOM property `document.location.href` gives the URL of the page in which it is evaluated. In order for an extension to interact with web pages through the DOM interface, it needs to have the permission to inject content scripts (Section 2). On the other hand, the Chrome extension API allows extensions to register callbacks for page loading events (e.g., `chrome.webRequest`), as well as to directly query for privacy-sensitive information (e.g., the `chrome.history` API gives access to the user's browsing history information). These APIs typically require permissions to be declared in the manifest files before they can be used.

Mystique's goal is to track the flow of data containing privacy-sensitive information inside third-party extensions. Therefore, it needs to mark the values obtained from these sources as tainted. Table 1 summarizes the taint sources considered by Mystique. We note that this is not intended to be an exhaustive list of all sources from which privacy-sensitive information can be obtained by extensions. Additionally, extensions can also leak privacy-sensitive information without having to access it *themselves*. For example, they can inject into the DOM of the current page an `img` element whose `src` attribute points to a third-party host and does not encode any tainted values, but the `Referer` field in the HTTP header of resulting request would be set by the browser and sent to the third-party host. This way, by reading the `Referer` filed of the incoming request, the third party can learn of the URL that the user is browsing while the extension that injected the element never accessed such information directly. However, this behavior is easy to detect at the network level since the `Referer` field is sent by the browser in plaintext, and although Mystique's taint tracking currently does not handle this scenario, it is nevertheless easy to modify Chromium so that any DOM element injected by extensions can be differentiated (whether it contains tainted values or not). For example, one way to achieve this is to hook the relevant DOM APIs to mark the elements when they are injected by extension JavaScript, which can itself be differentiated because the associated `Context` objects are different from those of normal website JavaScript (see Section 4.2 for explanation regarding the `Context` objects). We consider leakage via the `Referer` field out-of-scope for our current work since Mystique's focus is on tracking the usage of privacy-sensitive information inside extensions.

```
1  function encode_page_url() {
2    var loc = location.href;
3    var obj = { url: loc, length: loc.length };
4    var length = obj.length;
5
6    var output = "";
7    for (var i = 0; i < length; i++) {
8      var c = obj.url[i];
9      if (c == "a")
10       output += "a";
11     else if (c == "b")
12       output += "b";
13     /* repeated for all valid URL characters */
14   }
15
16   var result = window.btoa(output);
17   return result;
18 }
```

Listing 1: Sample JavaScript code, showing control dependency.

3.2 Taint Propagation with Static Analysis

To monitor the complete data flow of extensions, we not only need to consider all object types available in JavaScript, but also the conversion and interaction among different object types. For example, a `base64` encoding routine might first convert the input to integers, which are then used to index into a table to produce the output string. In this case, if the intermediate integers are not tainted, then we might fail to taint the output string. As mentioned, to be generic and avoid the complexities of JavaScript and the V8 engine, Mystique does not patch individual data flow operations to be taint aware, but rather leverages the fact that V8 has access to all the JavaScript source code that is to be executed, and propagates taint according to information obtained from a flow-insensitive, intra-procedural static analysis of the JavaScript source code.

Figure 2 illustrates an overview of this approach. The basic idea is to use the taint status of concrete runtime objects that the JavaScript code operates on (e.g., strings and numbers) to taint nodes (e.g., variables) in the abstract syntax tree (AST) parsed from the JavaScript source code. Taint propagation then starts from these tainted AST nodes by using a data flow graph (DFG) constructed from the AST. For each AST node that taint propagates to, their corresponding

Category	Taint source	Type	Requires permission?
DOM	`document.URL`	Property evaluation	Content script injection
DOM	`location`, `window.location`, `document.location`	Property evaluation	Content script injection
DOM	`document.cookie`	Property evaluation	Content script injection
DOM	`<input type="password">`	DOM query	Content script injection
Chrome Extension API	`chrome.tabs`	Event callbacks	"`tabs`" permission
Chrome Extension API	`chrome.webRequest`	Event callbacks	"`webRequest`" permission
Chrome Extension API	`chrome.webNavigation`	Event callbacks	"`webNavigation`" permission
Chrome Extension API	`chrome.history`	Direct query	"`history`" permission

Table 1: Taint sources considered by Mystique.

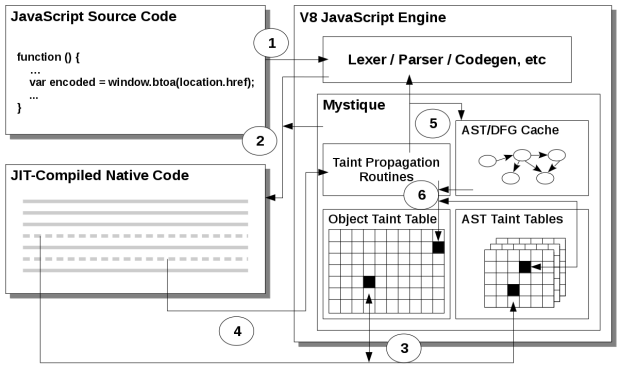

Figure 2: Overview of Mystique's approach to taint propagation. (1) and (2): JavaScript source code is compiled by V8 to native code and instrumented by Mystique; (3) During runtime, as expressions get evaluated, the instrumented native code looks up the evaluated values in the object taint table and updates the AST taint table; (4) At taint propagation points, the instrumented native code invokes Mystique to propagate taint; (5) Mystique reuses V8's existing infrastructure to parse for the function's AST and also constructs DFG, caching both; (6) Mystique propagates taint according to the DFG and AST taint table.

runtime objects are then also tainted. Mystique utilizes V8's existing infrastructure to parse the JavaScript source code for its AST. Mystique also collects control dependency information from the AST, and augments the DFG with this information (Figure 2). Listing 1 contains an example showing why control dependencies need to be taken into account: assuming `location.href` is tainted (and therefore also `obj.url` and `c`), if control dependencies are not considered, then the `if` branches starting from line 9 would allow an attacker to evade taint analysis (notice that string literals such as `"a"` are not tainted).

Mystique generates DFG in the unit of individual JavaScript functions, since V8 also parses for AST on a function-by-function basis. Note that the JavaScript top-level, or global, scope is treated as an unnamed function by V8. The V8 JavaScript engine does not currently cache but throws away the AST after the JavaScript source has been compiled. However, the JavaScript source code is always kept available, in order to support re-compilation by the optimizing compiler (as well as possible future de-optimization) [22]. To minimize performance overhead, Mystique caches the parsed AST and DFG (Figure 2).

To store taint data, two sets of tables are used by Mystique: one for storing taint status of concrete runtime JavaScript objects, and the other for tainted AST nodes (Figure 2). For convenience, we refer to the former as *object taint table*, and the latter as *AST taint table*. We need the latter since Mystique propagates taint based on the DFG, which is constructed from the AST. Note that there should be one AST taint table per function *invocation*, since each invocation (even of the same function) can operate on tainted values differently. This allows Mystique to handle taint propagation for recursive calls. During execution of the JavaScript code, the taint status of concrete runtime objects are looked up in the object taint table and used to update the AST taint tables.

Mystique triggers taint propagation for each individual JavaScript function at critical points during its execution. For example, since Mystique's static analysis is intra-procedural, it cannot readily track data flow across function boundaries. Therefore, taint propagation needs to be triggered on function calls so that if taint data should propagate to any of the callee's parameters (or any objects that the callee can access), it will be correctly reflected in the callee. This mechanism shown as the fourth step in Figure 2.

In the following, we present the details of Mystique's taint propagation approach.

3.2.1 Taint Representation for Runtime Objects. As mentioned, Mystique uses two sets of tables to store taint data. For the runtime JavaScript objects, their taint status is recorded in the object taint table, and we simply implement it as a global hash table that is keyed on the addresses of tainted runtime JavaScript objects. However, this simplistic scheme is complicated by the fact that V8 uses a garbage collector to recycle memory occupied by "dead" objects that are no longer accessible from the JavaScript code. Internally, the heap-allocated runtime JavaScript objects are always given word-aligned addresses, and V8 tags the pointers to heap-allocated objects by setting their lowest bit. This is done so that during garbage collection cycles these pointers can be distinguished from non-tagged (i.e., non-heap-allocated) values. There is only one type of non-tagged values used by V8, i.e., `Smi` for storing small integers (e.g., in the range $[-2^{30}, 2^{30} - 1]$) whose values fit in a word (minus the tag bit, since it is always cleared for untagged values). If the integer value is not in the given range, then it is stored in a normal heap-allocated object (i.e., `HeapNumber`, which is also used to store floating-point numbers). Note that no heap

memory is allocated to Smis - their values are stored directly in the non-tagged word. This arrangement by V8 leads to two problems: *a*) V8's garbage collector might move objects around in memory, so that the addresses stored in the object taint table become outdated, and *b*) if we taint Smis by storing their values inside the object taint table, then subsequent accesses to those same integer values will all be considered as tainted, leading to false positives.

Mystique addresses the first problem by making V8's garbage collector aware of the object taint table, i.e., if tainted objects are either moved in memory or freed, the object taint table is updated to reflect the change. Note that if tainted objects are freed, they can be safely deleted from the object taint table since they are no longer reachable from the JavaScript code. To solve the second problem, Mystique requires that the types of AST nodes that can be included in the DFG (e.g., variable nodes) can never refer to Smi values. Specifically, we modify V8's code generation phase so that assignments to such nodes are instrumented with checks to test if the values being assigned are Smis, and if so, replace them with equivalent (i.e., same numerical value) HeapNumbers. See Section 3.2.3 for AST node types that are included in the DFG. We remark that for function call nodes, since they are "assigned" when the callee returns, we also instrument return statements similarly so that Smi values are never returned.

3.2.2 Taint Representation for AST Nodes. Mystique stores taint status of AST nodes in the AST taint table, which is implemented as a hash table keyed on the memory addresses of tainted AST nodes (e.g., of a variable node). Notice that since V8 constructs AST at runtime, memory for AST nodes are always dynamically allocated. As mentioned, to account for different invocations of the same function, each invocation should be given its own table. Thus, to check the taint status of an AST node (of a function invocation), two levels of lookups are needed: starting from a global table, using in sequence *a*) the invocation's frame pointer, and *b*) the AST node's memory address as keys. It should be noted that in practice, most JavaScript functions never operate on tainted values, so the majority of lookups will not go beyond the first level. Also, for most of its operations involving the AST taint table, Mystique does not need to traverse the two levels of table lookups for *every* AST node. For example, when propagating taint for a function invocation, it would first get a reference to the second-level table, and all lookups then query this table directly.

Unlike runtime objects, memory allocated to the stack frames and the AST nodes is not managed by V8's garbage collector, so it does not need to be made aware of either of the two levels of tables.

3.2.3 DFG Generation. Mystique considers the JavaScript assignment operation as the primary means in which taint propagates, i.e., if the right-hand side expression of an assignment operates on tainted values, then the target on the left-hand side should also be tainted. To account for control dependencies (e.g., introduced by control structures such as while statements), Mystique treats their conditional expressions in the same way as the right-hand side of assignments, from which taint should propagate to the left-hand side of every assignment operation contained in that control structure (including nested statements). For control structures having more than one branch (e.g., if-else statements), taint propagates from the conditional expression to *all* branches. For convenience

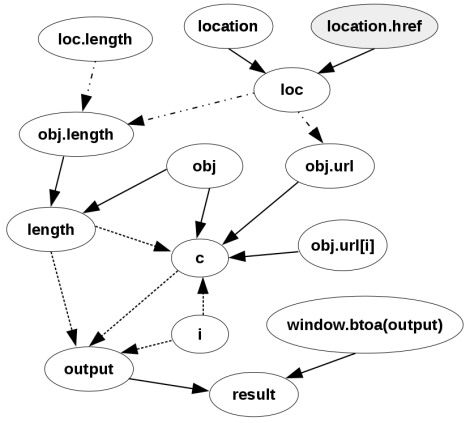

Figure 3: Data-flow graph (DFG) generated by Mystique for the code sample in Listing 1, augmented with control flow dependencies (dashed lines). Taint source (location.href) is highlighted. Also shown are implicit data flows (dashed-and-dotted lines), which are *not* encoded into the DFG.

we use the term *RHS expressions* to refer to both the right-hand side of assignments and conditional expressions of control structures.

RHS expressions are processed recursively, and when an AST node representing 1) variable, 2) object property, or 3) function call is encountered, an edge going from it to the left-hand side target (corresponding to the containing RHS expression) is added to the DFG. Note that array indexing (e.g., str[i]) is treated in JavaScript the same as object property access. We remark that if a function call is processed as part of an RHS expression, then Mystique not only generates a DFG edge from the call expression itself (representing the return value), but also recursively processes each argument expression as an RHS expression. Similarly, for object property accesses (e.g., obj.x), an edge is generated from the property itself, then the home object expression is also processed recursively. However, if the current RHS expression is an object or array literal, then Mystique does not process the expression. This is due to how Mystique handles tainting for objects and arrays (see Section 3.2.5).

Figure 3 shows the DFG, augmented with control dependency information, that Mystique generates for the example in Listing 1 according to the rules given above. We represent direct data flows as solid lines (e.g., from location.href to loc, corresponding to the assignment on line 2 in Listing 1), and control dependencies as dashed lines (e.g., from i to output). Note that the initialization of a variable in the var statement is treated in V8 as two separate operations: one as variable declaration and the other as an assignment with the initializer expression being the right-hand side. Figure 3 also shows implicit data flows (dashed-and-dotted lines), which we describe next, that are *not* generated as part of the DFG.

3.2.4 Implicit Data Flows. Besides explicit assignment operations and control dependencies, data flow paths can also be introduced implicitly in JavaScript. For example, at function calls, the actual parameters are themselves expressions which are evaluated and the resultant values "assigned" to the callee's formal parameters. In this case, the formal parameter should be treated in the same way

as the left-hand side of an assignment, and the actual parameter expression the right-hand side. Generally, implicit data flows are introduced whenever the evaluated value of an expression can be "caught" in some manner other than explicit assignment at the source level. For convenience we refer to this type of expressions as *implicit-flow expressions*.

Further examples of this include function return values and literals of compound types (e.g., object/array literals). For the former, if the return value expression is tainted, then the return value needs to be tainted; for the latter, if the expression that specifies the value of a property in the object literal is tainted, its corresponding property value should be tainted as well (Mystique does not taint the object literal itself, see Section 3.2.5). Figure 3 illustrates an example of implicit flow in the case of object literals (corresponding to line 3 in Listing 1), shown as dashed-and-dotted lines pointing from `loc` to `obj.url` and `obj.length`, and from `loc.length` to `obj.length`. Note that in this example, implicit flows are crucial in linking `location.href`, one of the taint sources considered by Mystique, to the final `result` variable (i.e., the return value of the function in Listing 1).

Mystique does not generate DFG edges for implicit data flows and their taint propagation is handled separately (discussed in Section 3.2.7).

3.2.5 Tainting of Object Properties. In contrast to previous approach (e.g., [47]), when tainting a specific property value, Mystique does not propagate taint to its containing object. For instance, if the property `obj.x` should be tainted, we only taint the property's AST node (and its corresponding runtime object), but we do not taint `obj`. This rule also applies to arrays, since array indexing is treated semantically in the same way as property access. However, note that due to how the DFG is constructed (Section 3.2.3), if an object is tainted, then all property accesses from that object will propagate taint from it (e.g., if `str` is tainted and `str.length` is assigned to a variable, then that variable will be tainted as well). Similarly, property accesses of a tainted object, if they constitute parts of an implicit-flow expression (Section 3.2.4), would also propagate taint to the evaluated value of that expression (Section 3.2.7 details how Mystique propagates taint for implicit-flow expressions).

3.2.6 Updating the AST Taint Tables. To accurately update the AST taint tables during runtime, we instrument the code emitted by V8 for the AST node types that can be included in the DFG. This instrumentation makes sure that as the expression represented by an AST node gets evaluated at runtime, the taint status of the resultant value is looked up in the object taint table and, if tainted, the AST node must also be tainted (i.e., it is recorded in the AST taint table). On the other hand, if the evaluated value is not tainted but the corresponding AST node is tainted, then the AST node should be untainted (this can potentially be the case for AST nodes belonging to loop statements). Doing so prevents overtainting and ultimately false positives. Note that if the value that an AST node refers to is replaced by a `HeapNumber` (as described in Section 3.2.1), this instrumentation will update the AST taint table using the replaced (i.e., `HeapNumber`) value.

3.2.7 Taint Propagation Points. Since Mystique adopts an intra-procedural static analysis, data flows across function boundaries are not reflected in the DFG. To solve this problem, Mystique requires that taint propagation be triggered on function calls and returns in order to update taint data for the callee and caller, respectively. Additionally, to more accurately capture the taint data flows in JavaScript, two more propagation points need be included in Mystique's analysis.

First, as mentioned in Section 3.2.6, to precisely model taint data flows in loop statements, Mystique allows untainting AST nodes when they no longer refer to tainted runtime objects. However, this can introduce inaccuracies into the analysis. For example, if a loop statement operated on tainted values during its execution, but on its last iteration it did not, then on exit from the loop statement none of the AST nodes belonging to it will be tainted. To address this scenario, Mystique requires that taint propagation be triggered at the end of a basic block.

Second, taint propagation should also be triggered when encountering implicit data flows (Section 3.2.4), so that if taint should propagate *to* the implicit-flow expressions, it is correctly reflected. Then, to handle taint propagation *from* implicit-flow expressions, Mystique treats them as RHS expressions: for all of the containing AST nodes from which a DFG edge should otherwise be generated (as described in Section 3.2.3), if any of them are tainted, then the evaluated values of the implicit-flow expressions need to be tainted as well (i.e., recorded in the object taint table).

3.2.8 Handling `eval` of Tainted Strings. Mystique handles `eval` of tainted strings in a manner similar to [47]: if a JavaScript function is compiled from a tainted string, Mystique then taints all left-hand side targets of assignment expressions in that function; for implicit-flow expressions, Mystique always taints their evaluated values.

3.3 Additional Data Flow Paths

Besides the data flow paths mentioned in Section 3.2, the Chromium browser also augments the semantics of JavaScript to implement both the DOM interface and its extension API, which creates additional data flow paths that are accessible to extensions. In the following we detail how Mystique propagates taint across these paths. Note that the data flow paths mentioned in this section are those that are currently considered by Mystique's analysis, and should not be considered as an exhaustive list of data flow paths accessible to extensions through the DOM or the extension API.

3.3.1 The DOM Interface. The DOM interface is not implemented by V8. Instead, it is implemented as an add-on to JavaScript in WebKit/Blink [2]. Since the DOM implementation is external to V8, JavaScript values written to DOM need to be converted to their corresponding representations in WebKit/Blink (and vice versa when values are read from the DOM). Extensions that have declared the permission to inject content scripts can use them to interact with the DOM interface of web pages. For example, they can invoke the `setAttribute` method on an element in the DOM, and later read back that attribute's value. Given the value conversion between V8 and WebKit/Blink, the read-back value may be a separate copy of the original and therefore not tainted. To solve this problem, Mystique also taints in WebKit/Blink any values that are tainted in V8. When such tainted values are read by JavaScript, Mystique ensures that they are tainted in V8.

We remark that as an implementation choice, our current Mystique prototype (see Section 4) does not cover string manipulations (e.g., concatenation) that are *internal* to WebKit/Blink, nor do we handle the HTML tokenizer that WebKit/Blink invokes to parse HTML content written to the DOM via JavaScript (e.g., by writing to `innerHTML`). We stress that these are particular implementation choices and not a fundamental limitation of Mystique's methodology. Previous work [36] has already made the internal operations of WebKit/Blink taint-aware.

3.3.2 Chrome Extension API. As mentioned in Section 2, the Chromium browser provides an API for message passing between content scripts, which execute inside of web pages, and the rest of the extension (e.g., the background page) [11]. This API allows JavaScript objects to be passed through it. Internally, the objects being sent are first serialized (using `JSON.stringify`) and then given to the (C++) IPC implementation to be delivered to the receiving end of the message pipe, where the original objects are then parsed back. However, if the original objects contain tainted values, then the objects parsed back need to reflect the same taint status. To minimize changes to the code base, we chose not to make the IPC implementation taint-aware. Instead, for each JavaScript object sent through the message passing API, Mystique visits each of its properties recursively and constructs a "meta-object" that describes its taint status. This meta-object is then also stringified, and the result is then prepended to the stringified output of the original JavaScript object. On the receiving end, the meta-object is used to reconstruct the taint status of the parsed back object.

Another data flow path introduced by the extension API is the `executeScript` method in `chrome.tabs` [5], which as mentioned allows programmatic injection of strings that execute as content scripts in web pages. If the injected scripts are tainted, we treat them in the same way as `eval` of tainted strings (see Section 3.2.8).

Finally, Chromium provides an API (`chrome.storage`) that allows extensions to serialize JavaScript objects to storage [4]. Similar to the messaging API, serialization is done using `JSON.stringify`. Therefore, Mystique also handles taint propagation for it by use of meta-objects that are serialized together with the original objects.

3.4 Taint Sinks

To detect extensions that abuse their privileges to gather and exfiltrate users' privacy-sensitive information, Mystique currently considers the following as taint sinks in its analysis:

- `XMLHttpRequest`: An alert is raised if tainted values form any part of the request URL parameter or the request body.
- `WebSocket`: Similar to `XMLHttpRequest`, an alert is raised if any tainted values are sent via this interface.
- `chrome.storage`: Raise an alert if tainted values are given to the `chrome.storage` API for persistence.
- For the DOM elements injected by extensions, raise an alert if their `src` attributes contain tainted values.

We consider the `chrome.storage` API to be a taint sink, for the case where an extension stores privacy-sensitive information first and only exfiltrates them later in bulk (e.g., only after collecting a threshold number of URL visits). Note that as with all dynamic analysis systems, Mystique might not be able to generate the threshold number of events at runtime in order to trigger the leaking behaviors for an extension. In such cases, Mystique will still be able to flag the extension if it uses the `chrome.storage` API to persist data across runs. Previous approaches (e.g., [43, 52]) that rely solely on observing the network traffic generated by an extension would fail to detect such cases.

DOM elements injected by extensions are also considered as taint sinks, since for such elements the browser will try to fetch their content from the URL specified in the `src` attribute. Thus, extensions can leak sensitive information by, for example, encoding it as part of the `src` attribute URL.

3.5 Taint Propagation Logs and Sink Report

To keep track of tainted data flows inside extensions, each step of taint propagation needs to be logged. In its basic form, for each tainted JavaScript object, Mystique logs the JavaScript object from which taint propagated to it, along with the JavaScript function and source code position in that function where propagation occurred. For JavaScript objects that are taint sources, whose taint did not propagate from another object (e.g., `document.location.href`), Mystique logs them in a separate table along with the JavaScript function and position inside that function where they are accessed.

In cases where taint propagation is due an meta-object (see Section 3.3.2), Mystique first dumps on the filesystem the propagation logs of all the tainted JavaScript values that the meta-object describes. Then, the filesystem paths of the propagation logs are encoded inside the meta-object. These filesystem paths would then be used to indicate the previous step in taint propagation once the original object is parsed back.

For JavaScript functions that are compiled from tainted strings (due to either `eval` or the extension API's `executeScript` method), as mentioned Mystique taints all left-hand side targets of assignments as well as the evaluated values of implicit-flow expressions. For these values, the previous step in taint propagation would be the tainted string from which the function was compiled.

When tainted values reach taint sinks, Mystique logs the event by recording 1) the tainted values that triggered the taint sink and their propagation logs, 2) the current JavaScript stack trace, and 3) the source code of JavaScript functions along the propagation paths (including the functions that accessed taint sources).

4 IMPLEMENTATION

We implemented a prototype of Mystique for the Chromium browser. Apart from our description in Section 3, we detail in the rest of this section the additional changes that our prototype implementation added to V8 and WebKit/Blink.

4.1 Mapping AST Nodes to JavaScript Objects

Given Mystique's approach to taint propagation detailed in Section 3, it is necessary to know, for each AST node that taint propagates to, what JavaScript object they referred to *at the instant* when their values were evaluated during runtime. This is needed so that the object taint table can be updated correctly. To obtain this mapping information (from AST nodes to JavaScript objects), we used a method similar to that described in Section 3.2.6, namely, for each AST node that can be included in the DFG, we instrument the

native code emitted for it. This instrumentation makes sure that the evaluated value of the expression corresponding to the AST node is recorded in a table. We refer to this table as the *AST-to-object mappings table*. Note that as with the AST taint table, there needs to be one AST-to-object mappings table per function invocation.

4.2 Optimizing Taint Propagation

V8 uses a `Context` object [8] to model an execution environment that corresponds to a global variable scope in JavaScript. To minimize runtime overhead incurred by taint propagation, in our prototype implementation of Mystique we chose to only propagate taint for JavaScript code with a `Context` (i.e., global variable scope) belonging to extensions. This optimization is sound since JavaScript code with a given global variable scope cannot access objects defined in another. Furthermore, even though content scripts are injected and run in the "context" web pages, they cannot use variables or functions defined by 1) the web pages, 2) other content scripts, or even 3) their own extension's pages [6]; internally, this rule is enforced by defining separate `Context` objects for content scripts. Lastly, JavaScript code from different extensions is given separate `Contexts`.

Although the `Context` class is defined and implemented in V8, their runtime instantiation is initiated by WebKit/Blink, which differentiates the `Contexts` of web pages and extension content scripts by the terms *main* worlds and *isolated* worlds, respectively [13]. That is, the content scripts of an extension are run inside isolated worlds. To implement our optimization of propagating taint only for extension `Contexts`, we modified WebKit/Blink so that it notifies Mystique whenever a `Context` is instantiated for isolated worlds. On the other hand, the background page of an extension is treated by WebKit/Blink as a normal web page, in the way that the JavaScript code of the background page is run inside the main world. Thus, we also need to have WebKit/Blink notify Mystique when a `Context` is instantiated for a background page in the main world. In our prototype implementation, we currently identify background pages by checking if the page URL begins with "`chrome-extension://`".

We note that for JavaScript code with a non-extension `Context`, it is also not necessary to: 1) taint any values obtained from the taint sources described in Section 3.1, and 2) maintain the corresponding AST taint table or the AST-to-object mappings table.

4.3 `JSON.stringify` and `JSON.parse`

Evidently, if the JavaScript object passed to `JSON.stringify` contains tainted properties, then the output string needs to be tainted as well. However, given how Mystique handles tainting of object properties (Section 3.2.5), if the input object *itself* is not tainted but nevertheless contains property values that are tainted, then taint would not propagate to the output string. This is due to the fact that V8 implements `JSON.stringify` as a built-in function directly in C++. To propagate taint, we added a JavaScript trampoline function that calls the underlying C++ implementation. This trampoline is also responsible for constructing a meta-object (see Section 3.3.2) that describes the taint status of the object being serialized. If the input to `JSON.stringify` contains tainted values, the trampoline function will taint the output, as well as update the propagation logs using the meta-object as the previous step in taint propagation.

Given our changes to `JSON.stringify`, if the input string to `JSON.parse` is tainted, we have two possibilities: 1) the string's previous step in taint propagation is a meta-object, in which case Mystique reconstructs taint according to the meta-object, otherwise 2) Mystique sets taint for the output object and recursively for all of its properties. As with `JSON.stringify`, we also inserted a JavaScript trampoline function for `JSON.parse` to achieve these.

Our treatment of `JSON.stringify` and `JSON.parse` is similar to how we handle the Chrome message passing and storage APIs (Section 3.3.2). However, note the difference here is that the output of `JSON.stringify` and `JSON.parse` stays within the V8 heap memory, instead of being given to the C++ IPC implementation or to external storage.

	# Extensions	# Flagged	Percentage
Chrome	178,893	3,809	2.13%
Opera	2,790	59	2.11%
Total	181,683	3,868	2.13%

Table 2: Summary of dataset and analysis results.

4.4 jQuery Request Protocol

The jQuery library is frequently used by third-party Chrome extensions. To support protocol-less URLs (which start with "`//`"), the jQuery library first retrieves the current page's URL by reading the DOM property `location.href` and parses for its protocol (e.g. `https`), which is then prepended to the request URL if its protocol is not specified [10]. The request URL is later passed to the `open` method of `XMLHttpRequest`. Since `location.href` is treated by Mystique as a taint source and `XMLHttpRequest`'s request URL as taint sink, this will cause an alarm to be triggered regardless of whether the original request URL is tainted or not. To fix this false positive case, Mystique creates a signature based on the AST structure of the particular assignment expression that is responsible for falsely propagating taint (i.e., [10]), and does not propagate taint for the expression when it is encountered.

5 EVALUATION

In this section, we describe our experimental setup and present the results of applying Mystique to large-scale analysis of browser extensions. The dataset used in the analysis include 178,893 extensions that were crawled from the Chrome Web Store between September 2016 and March 2018, as well as a separate set of all available extensions (2,790 in total) for the Opera browser at the time of our analysis. The Chrome extensions in our dataset typically contain multiple versions of the same extension. Discounting version differences, our dataset contains 118,083 unique Chrome extensions. Our analysis loads the Opera extensions in our modified version of Chromium - this works in most cases since the Opera browser is based on Chromium [49]. We were able to analyze all of the 2,790 Opera extensions using this method. As shown in Table 2, Mystique flagged 3,868 Chrome extensions and 59 Opera extensions as potentially leaking privacy-sensitive information. The total number of flagged extensions is 3,868 (2.13%). We finished analyzing all of

the 181,683 extensions in less than a month with our experimental setup, as described next.

5.1 Experimental Setup

Our analysis is automated using Selenium's ChromeDriver [3]. For each extension, we launch a fresh instance of our modified Chromium with the extension pre-installed using Chromium's `--load-extension` command line argument. We then simulate web browsing by using Selenium to drive the Chromium browser to visit a fixed set of URLs.

We divide this fixed set of URLs that we use to simulate web browsing into two categories: *real URLs* and *mock URLs*. For the real URLs, we serve real website responses; for the mock URLs, we serve only a static mock page. The motivation for using mock pages is to shorten analysis time by avoiding spending the time needed to load a real page inside the browser, while still be able to generate many URL load events for the extension being analyzed. In total we have 10 real URLs (of popular websites such as wikipedia.org). Mock URLs are derived from the real URLs: for each real URL, we visit it in the browser and randomly select 10 URLs that it links to (the selection was done programmatically by using Python's `random` module). Thus, in total, we drive the Chromium browser to visit 110 URLs for each extension analyzed.

Using mock pages is a common methodology that was adopted in similar works that dynamically analyze browser extensions (e.g. [43]). However, one potential shortcoming of this is that an extension might expect the web page to have specific structural layouts (e.g., certain DOM elements need to be present) before it manifests malicious behaviors. To elicit malicious behaviors from such extensions, we incorporated HoneyPage [34] into our analysis whenever mock URLs are visited. Note that for real URLs, it is not necessary to use HoneyPage since the real website responses are served in this case.

To minimize network traffic as well as to make the analysis more reproducible, we used a tool [19] to implement a replay cache that serves pre-recorded responses for real URLs. We pre-record the responses for a real URL by visiting that URL and using the same tool to save the website responses into the replay cache. We also modified the tool so that when a response is not found in the replay cache, it fetches the latest content from the Internet instead of the default behavior of returning an error status. This modification is needed since we don't want to restrict an extension's network access (e.g., extensions might query a remote server for configuration files, which is not pre-recorded in the replay cache). We remark that for dynamic websites (e.g., amazon.com), their responses are non-deterministic so that we cannot pre-record all of the possible responses (i.e., there will likely be cache misses for such websites).

We use `mitmproxy` [12] to serve mock page for the mock URLs, as well as to log network traffic generated during the analysis. We arranged the replay cache and `mitmproxy` in a way such that contents already pre-recorded in the replay cache will not be logged again by `mitmproxy`.

The analysis infrastructure used in our experimental evaluation consists of a local Kubernetes [35] cluster that can run 120 threads simultaneously. We used this cluster for parallel processing of the extensions. Each instance of our modified Chromium browser is run in a separate Docker [7] container.

	Sample Size	# TP (%)	# FP (%)	Precision
Chrome	349	272 (77.94%)	30 (8.60%)	90.07%
Opera	59	45 (76.27%)	6 (10.17%)	88.24%
Total	408	317 (77.70%)	36 (8.82%)	89.80%

Table 3: Quantifying true positive rates (TP = True Positive, FP = False Positive). Numbers in the "Precision" column are calculated as TP/(TP + FP).

5.2 Quantifying True Positive Rates

We attempt to quantify the true positive rate of Mystique's analysis by manual verification of the flagged extensions. Specifically, for each flagged extension, we examine their taint sink objects to see if they contain privacy-sensitive information and/or any data derived from such information (e.g., encrypted/hashed). We manually verified all of the 59 flagged extensions for the Opera browser. To estimate the true positive rate for the Chrome extensions, we manually verified a randomly selected sample of 349 extensions (out of a total of 3,809 flagged Chrome extensions). This sample size was chosen to target a confidence interval of 5% at a 95% confidence level, according to the standard theory on confidence intervals for proportions (e.g., [18], Chapter 13).

Most of the taint sink objects encountered during this process are in plaintext. For the taint sink objects that are apparently obfuscated, encrypted and/or cryptographically hashed, we confirm by examining the relevant portions of the extension's source code (given by the propagation logs as described in (Section 3.5)). Table 3 summarizes the results. We confirmed 272 and 45 true positives (TP) for the Chrome and Opera extensions, respectively. We were also able to confirm 30 and 6 false positives (FP) for the Chrome and Opera extensions, respectively. Note that we were not able to, from examining the taint sink objects, definitively confirm the true/false positive status for 47 of the Chrome extensions in our sample and 8 of the Opera extensions. Their taint sink objects do not seem to indicate the use of obfuscation, encryption and/or hashing. To ascertain their true/false positive status would require us to manually go through their *entire* propagation logs one by one, a resource expenditure that we currently do not have. We give our best effort estimate of the actual true positive rate in the "Precision" column in Table 3, calculated as TP/(TP + FP).

Since Mystique propagates taint across control-flow dependencies in addition to tracking direct data flows (Section 3.2.3), which as we mentioned is necessary for detecting privacy leakage from extensions, some false positives in the analysis result are inevitable. During the development of Mystique we did investigate some false positive extensions and found overtainting due to control-flow dependencies. However, verifying that this is indeed the cause of all the false positives would again require us to manually examine the taint propagation logs one by one, which as we mentioned is beyond our current resource expenditure. To ascertain the reason for the false positives, further improvements to Mystique's taint propagation logs should 1) label the taint data propagated from control-flow dependencies differently than normal data flows, and 2) include more detailed information to facilitate easy pin-pointing

of where a control-flow dependency originates. These improvements would also help a human analyst to quickly discern false positive results, mitigating the inaccuracy introduced by control-flow dependencies. Nevertheless, our prototype implementation of Mystique already provides reasonable accuracy for detection of privacy-leaking extensions, and we argue that in its current form, Mystique can be incorporated as part of a triage system for online extension repositories such as the Chrome Web Store.

5.3 Case Studies

In this section we present the details of Mystique's analysis results, highlighting the privacy threats posed by third-party extensions.

5.3.1 Number of Affected Users. We begin by capturing the number of users that are affected by extensions that were flagged by Mystique. These are extensions that were found and/or have the potential to leak users' privacy-sensitive information. To do this, we first sort all of the 3,809 flagged Chrome extensions according to the number of users they have. Starting from the extension with the most users, we then manually verify their taint sink objects to see if they are true positive results, as we have done in Section 5.2. Table 4 shows the number of users for the top 10 true positive Chrome extensions that were flagged by Mystique (10,000,000+ means more than 10 million users). The data on the number of users were obtained by crawling the Chrome Web Store after Mystique has finished analyzing the extension.

From the top 10 extensions in Table 4, 7 used `XMLHttpRequest` directly to send privacy-sensitive information to third-party, while the remaining 3 used the `chrome.storage` API to persist such information. Note that all `XMLHttpRequests` made by the extensions listed in Table 4 are sent to remote network hosts (instead of `localhost`, see Section 5.3.4). Although third-parties cannot immediately learn of privacy-sensitive information that was persisted using the `chrome.storage` API, extensions that stored tainted values with this API should still be labeled suspicious and further investigated, for the reasons that 1) the extension might be bulk-sending such information and our analysis did not trigger the conditions necessary for the actual leaking behavior, and 2) an updated version of the extension may leak the stored information, even though the current version only stores the information for local use. The latter is especially problematic when the original author decides to sell her extension to another entity, or when her account was hijacked by a malicious actor to push out bogus updates.

It should be noted that many of the Chrome extensions flagged by Mystique are no longer available on the Chrome Web Store (either taken down by Google or the extension authors), and therefore we were not able to collect data on the number of users for these extensions. There are 1,084 unique extensions (i.e., discounting version differences) in the 3,809 Chrome extensions flagged by Mystique. Out of these 1,084 unique extensions, we obtained data on the number of users for 659 of them.

In the rest of this section, we give case studies of representative extensions that were detected by Mystique. We also show the strength of Mystique by comparing it with similar previous efforts.

5.3.2 SimilarWeb Library and Web of Trust. Here we look at Mystique's effectiveness at flagging extensions that have been known to leak privacy-sensitive information. The SimilarWeb tracking library and the Web of Trust extension are two cases that have drawn attention from the security community in the past. In particular, The SimilarWeb library was identified in [51], and it is often bundled by extension developers to serve as a revenue source. In the report [51], the author was able to find 42 extensions out of the 7,000 most popular extensions on the Chrome Web Store by searching in the extension package for specific source code strings. In comparison, Mystique found a total of 382 extensions (or 99 unique extensions discounting version differences) that include the SimilarWeb library. *Note that all of the 382 extensions were detected by Mystique's automatic analysis. Our emphasis here is to highlight Mystique's ability to detect additional cases that were missed by the approach in [51].*

Specifically, to find extensions that contain the SimilarWeb library from Mystique's analysis results, we filtered the extensions flagged by Mystique (3,868 in total) by similarity in the general format of the taint sink objects. We further verified that they indeed executed the SimilarWeb library by manually examining the taint propagation logs generated by Mystique (Section 3.5). Overall, we were able to identify 5 more domains that received the leaked data than the original report [51] (these are: starwebnet.com, fvdsuggestions.com, upgit.com, analyticstats.com, and connectwebonline.net). We also found extensions that 1) does not package the SimilarWeb library code directly but nevertheless download and execute it at runtime, and/or 2) use minified versions of the library. Note that this highlights the drawbacks of the approach used in [51]: it cannot catch extensions that load tracking code externally at runtime, neither can it catch minified/obfuscated cases, since it only searched in the extension package for specific source code strings.

We next turn our attention to Web of Trust (WOT) [20], an extension that provides its users with reputation and safety information about the websites based on popular reviews. Note that in order to provide its advertised functionalities, WOT has to collect the current browsing activity and use it to query a remote server. However, as mentioned, WOT has reportedly been selling its users' browsing history to third parties [41], which demonstrates the potential threat of privacy abuse by such extensions.

Mystique was able to flag the WOT extension as leaking privacy-sensitive information. By examining the taint propagation report, we found that the WOT extension obfuscates the leaked data by first applying RC4 encryption and then encoding the result with double-base64 (i.e., twice base64-encode). Additionally, Mystique also flagged another extension, Filter by WOT, released by the same developers as WOT, that demonstrates the same behavior. Since these two extensions use encryption, the method used by Starov et al. [43] cannot use its heuristics to decrypt the data, and therefore would not be able to detect them.

5.3.3 Comparison with Traffic Analysis Methods. Previous research efforts at identifying privacy-leaking browser extensions using dynamic analysis focused only on the network traffic generated by extensions (e.g., [43]). Since they lack the insights into the detailed data flows inside the extensions, they have to rely on heuristics to attempt de-obfuscation of any encoded parameters and recover the plaintext. As such, they cannot handle 1) encoding schemes that

	# Users	Taint Sink(s) Triggered
Avast SafePrice	10,000,000+	`chrome.storage`
Avira Browser Safety	10,000,000+	`XMLHttpRequest`
Avast Online Security	10,000,000+	`chrome.storage`
Pinterest Save Button	10,000,000+	`XMLHttpRequest`
Unlimited Free VPN - Hola	8775275	`XMLHttpRequest`
AVG SafePrice	5585975	`chrome.storage`
Pop up blocker for Chrome™- Poper Blocker	2292266	`XMLHttpRequest`
Block Site - Website Blocker for Chrome™	1468846	`XMLHttpRequest`
Trustnav Safesearch	1340990	`XMLHttpRequest`
WOT Web of Trust, Website Reputation Ratings	1231219	`XMLHttpRequest`

Table 4: Top 10 true positive extensions flagged by Mystique.

are not anticipated beforehand, and 2) more importantly, extensions that use one-way hashing (e.g., MD5) or encryption. Furthermore, they cannot identify cases where extensions persist privacy-sensitive information on local storage, which as we mentioned in Section 5.3.1 presents opportunities for abuse and should be labeled for further investigation. Finally, since Mystique logs each step of taint propagation, it is also able to give detailed information on how sensitive data are abused *within* third-party extensions, an insight that pure network-level analysis lacks.

We present in this section case studies of extensions that were detected by Mystique, but would have been missed by previous traffic analysis methods. We start by giving two example encoding schemes that we found through Mystique, but would have evaded the heuristics used by Starov et al. [43]. The first one, which we term string-to-hex encoding, is a method that simply converts each character to a two-digit hexadecimal number according to its integer value in the ASCII table. For example, in this scheme, the string "abc" would be encoded as "616263" (0x61 is the ASCII value for the character 'a'). The second encoding scheme uses plain base64 encoding, but appends to the end a fixed string `"/version=2.x/*"` (`x` ranges from 0 to 2 in our detected extensions). Since the Starov et al. [43] only considered URL encoding, base64, repeated base64, gzip/deflate, and JSON-packing to attempt decoding of each individual parameters, their methodology cannot detect the string-to-hex encoding scheme. For the second scheme, although plaintext can still be recovered by attempting unmodified base64 decoding, this would not be the case if the version string was *prepended* rather than appended (i.e., slight modifications to the standard encoding schemes would invalidate their heuristics). The simplicity of these two encoding schemes nevertheless points to the problems faced by approaches that rely on heuristics to decode network traffic.

On the other hand, if the extension employs one-way hashing or encryption, then it is not possible to recover the plaintext by merely applying heuristic to the captured network traffic. For one-way hashing, third-parties that are interested in learning the browsing activities of the extension users can simply build beforehand a table of hashes of all the popular website's URLs (e.g., from Alexa top sites). The most frequently encountered hash algorithm used by extensions is currently MD5. For extensions that use encryption, apart from the aforementioned WOT extension, which uses RC4, we also observed another extension that uses ROT-13 (a simple encryption that maps a letter to another that is 13 places after it in the alphabet).

We point out that although traffic analysis methods that match the traffic features to those generated by privacy-leaking extensions (e.g. [52]) can potentially catch extensions that use encoding methods not thought of beforehand, or even those that use encryption, such methods are prone to evasion whereby attackers mask their network traffic with noise. Mystique's tracking of tainted data flows is not affected by such evasion.

5.3.4 Extensions that leak to `localhost`. From Mystique's analysis results, we frequently find extensions whose taint sink objects are seemingly sent to `localhost` on a particular port. Upon closer examination, these extensions typically serve as a complement to native applications that are already installed on the user's computer. These native applications are the ones responsible for listening for the requests on `localhost`. It is possible that the desktop application would then be the one that actually leaks browsing activities to a remote server. To be conservative, any such extension should be labeled for further investigation (i.e., leakage to `localhost` should be treated in the same way as triggering of local storage taint sink).

6 LIMITATIONS AND FUTURE WORK

Mystique fundamentally relies on runtime dynamic analysis of browser extensions in order to detect privacy-intrusive behaviors, and as with all dynamic analysis systems, the successful detection of malicious behaviors depends on triggering such behaviors during the analysis. And even though we incorporated HoneyPage [34] into our analysis to aid in actively triggering malicious behaviors from extensions, HoneyPage is not without limitations and cannot guarantee complete coverage. While code coverage is an important metric to obtain for a dynamic analysis system such as Mystique, its measurement poses challenges in the web context. For example, a browser extension can implement some part of its functionality in JavaScript code that is fetched remotely from a server during runtime, and the fetched content might be different depending on a number of factors, such as whether the server is being queried by a security crawler (i.e., web cloaking [32]). To this end, our future research efforts will look into complementing Mystique with static program analysis techniques, such as [30, 42], in order to deduce properties of the JavaScript source code and automatically trigger privacy-intrusive behaviors. It will also be helpful to combine

techniques from previous works to detect when when cloaking is employed by malicious sites.

Privacy-sensitive information can also be leaked via side-channel attacks (e.g., [50]). In this paper, along with other similar previous research efforts, we consider side-channel attacks to be out-of-scope. Orthogonal techniques are needed to mitigate the impact of side-channel attacks.

7 RELATED WORK

Information flows in the web context: A number of previous works have investigated applying taint analysis to the web security context. Specifically, Lekies et al. [36], Stock et al. [45], and Melicher et al. [38] detect DOM-based cross-site scripting (XSS) vulnerabilities by augmenting both the V8 JavaScript engine and WebKit to make Chrome taint-aware. As mentioned, their system only handles direct taint propagation between strings, and as such is not adequate in detecting privacy-leaking extensions. Earlier works [25, 47] have implemented taint tracking for the Firefox browser by extending the SpiderMonkey [40] JavaScript engine. Although these were able to cover all available object types in JavaScript as well as control-flow dependencies, they relied on instrumenting all (bytecode) data flow operations emitted by SpiderMonkey. This approach is not suitable for our purposes, given the complex nature of the V8 engine as mentioned in Section 3.

The taint analysis technique that Mystique leverages fits in a broader category of works that deal with information flows in the web security context. Jang et al. [33] use source-level rewriting to track information flows in JavaScript web applications. However, they pointed out that their approach did not cover browser built-in APIs, nor did they handle data flows through the DOM, both of which are handled by Mystique. Also, their approach introduces source-level changes (e.g., script size) that might alter program behaviors. Similar rewriting-based approach was also used by Chudnov et al. [21], who propose an information flow control (IFC) monitor, but substantial additional work is needed for their approach to work with existing JavaScript applications. Another work [23] implements IFC for Firefox to enforce confidentiality policies between web scripts and browser APIs. Finally, Bauer et al. [17] offer an approach to enforce coarse-grained information flow policies among entities in the browser (e.g., DOM elements, events, and extensions), but it lacked Mystique's ability to track the detailed data flows within JavaScript applications (they treated the V8 JavaScript engine as a black box in their analysis).

As a complement to dynamic analysis techniques, previous works (e.g., [15, 30, 42]) applied static program analysis techniques in the web context to secure information flows, and/or to provide other security properties (e.g., Saxena et al. [42] implemented symbolic execution for JavaScript to automatically find vulnerabilities).

Detecting privacy-leaking extensions: Previous research [28, 37, 43, 52] also dealt with the privacy implications of browser extensions. In particular, Starov et al. [43] and Weissbacher et al. [52] are closely related to Mystique. Mystique differentiates in methodology from these prior works. As we mentioned the approaches adopted by Starov et al. [43] and Weissbacher et al. [52] rely on monitoring and analyzing network traffic, and we already compared Mystique with them in regard to methodologies in Section 5.3.3. Note that unlike Mystique, Starov et al. [43] and Weissbacher et al. [52] only analyzed the top 10K most popular extensions on the Chrome Web Store.

Large-scale studies of browser extensions: Apart from some of the works mentioned above (e.g., [15, 43]) and Mystique, researchers have also proposed a number of dynamic analysis frameworks aimed at automatically analyzing large numbers of browser extensions for malicious behaviors. Kapravelos [34] actively elicit malicious behaviors from browser extensions by developing HoneyPages, which we incorporated into our evaluation of Mystique. Another work [48] uses an instrumented Firefox browser to automatically analyze extensions for dangerous behaviors.

8 CONCLUSION

In this paper, we presented the design and implementation of the first information flow tracking tool for the Chromium browser. To overcome the complexities posed by both the JavaScript language as well as the V8 engine, we adopted a novel hybrid approach to runtime taint propagation. Based on this tool, we proposed Mystique, a framework for analyzing Chrome extensions. We applied Mystique to conduct a large-scale study of extensions from the Chrome Web Store, with respect to their privacy practices. To this end, we analyzed 178,893 Chrome extensions and 2,790 Opera extensions, and flagged 3,868 (2.13%) of them as potentially leaking privacy-sensitive information. We found that the top 10 of the Chrome extensions that we confirmed to be leaking privacy-sensitive information have more than 60 million users combined. We also uncovered a number of obfuscation methods that were missed by previous work. Our results demonstrate the feasibility and effectiveness of Mystique, and shed light on the privacy practices of browser extensions, highlighting the threat posed to user privacy.

9 ACKNOWLEDGMENTS

We would like to thank our shepherd Yinzhi Cao for his insightful comments and feedback. This work was supported by the National Science Foundation under grants 1703375 and by the Office of Naval Research (ONR) under grant N00014-17-S-B001.

REFERENCES

[1] 2018. Background Pages. https://developer.chrome.com/extensions/background_pages.
[2] 2018. Blink. https://www.chromium.org/blink.
[3] 2018. ChromeDriver - WebDriver for Chrome. https://sites.google.com/a/chromium.org/chromedriver/.
[4] 2018. chrome.storage. https://developer.chrome.com/extensions/storage.
[5] 2018. chrome.tabs. https://developer.chrome.com/extensions/tabs.
[6] 2018. Content Scripts. https://developer.chrome.com/extensions/content_scripts.
[7] 2018. Docker. https://www.docker.com/.
[8] 2018. Embedder's Guide. https://github.com/v8/v8/wiki/Embedder's-Guide#contexts.
[9] 2018. JavaScript APIs. https://developer.chrome.com/extensions/api_index.
[10] 2018. jQuery Prepending Protocol. https://github.com/jquery/jquery/blob/2d4f53416e5f74fa98e0c1d66b6f3c285a12f0ce/test/data/jquery-1.9.1.js#L8088.
[11] 2018. Message Passing. https://developer.chrome.com/extensions/messaging.
[12] 2018. mitmproxy. https://mitmproxy.org/.
[13] 2018. V8BindingDesign.md. https://cs.chromium.org/chromium/src/third_party/WebKit/Source/bindings/core/v8/V8BindingDesign.md.
[14] 2018. What are extensions? https://developer.chrome.com/extensions.
[15] Sruthi Bandhakavi, Samuel T King, Parthasarathy Madhusudan, and Marianne Winslett. 2010. VEX: Vetting Browser Extensions for Security Vulnerabilities.. In *USENIX Security Symposium*, Vol. 10. 339–354.
[16] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. 2010. Protecting Browsers from Extension Vulnerabilities.. In *NDSS*.

[17] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium.. In *NDSS*.

[18] Per Nikolaj D Bukh. 1992. The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling.

[19] Catapult Project. 2018. Web Page Replay. https://github.com/catapult-project/catapult/tree/master/web_page_replay_go.

[20] Chrome Web Store. 2018. Web of Trust. https://chrome.google.com/webstore/detail/wot-web-of-trust-website/bhmmomiinigofkjcapegjjndpbikblnp?hl=en-US.

[21] Andrey Chudnov and David A Naumann. 2015. Inlined information flow monitoring for JavaScript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 629–643.

[22] Jay Conrod. 2018. A tour of V8: Crankshaft, the optimizing compiler. http://www.jayconrod.com/posts/54/a-tour-of-v8-crankshaft-the-optimizing-compiler.

[23] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 748–759.

[24] Detectify Labs. 2018. Chrome Extensions - AKA Total Absence of Privacy. https://labs.detectify.com/2015/11/19/chrome-extensions-aka-total-absence-of-privacy/.

[25] Mohan Dhawan and Vinod Ganapathy. 2009. Analyzing information flow in JavaScript-based browser extensions. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 382–391.

[26] Vladan Djeric and Ashvin Goel. 2010. Securing script-based extensibility in web browsers. In *Proceedings of the USENIX Security Symposium*.

[27] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2010. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*.

[28] Cristiano Giuffrida, Stefano Ortolani, and Bruno Crispo. 2012. Memoirs of a browser: A cross-browser detection model for privacy-breaching extensions. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM, 10–11.

[29] Google. 2018. Chrome V8. https://developers.google.com/v8/.

[30] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. 2011. Verified security for browser extensions. In *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 115–130.

[31] How-to Geek. 2018. Warning: Your Browser Extensions Are Spying On You. https://www.howtogeek.com/180175/warning-your-browser-extensions-are-spying-on-you/.

[32] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean-Michel Picod, and Elie Bursztein. 2016. Cloak of visibility: detecting when machines browse a different web. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 743–758.

[33] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In *Proc of the 17th ACM Conf on Computer and Communications Security CCS*, Vol. 10. 43–51.

[34] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14)*. 641–654.

[35] Kubernetes. 2018. Production-Grade Container Orchestration. https://kubernetes.io/.

[36] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 Million Flows Later - Large-scale Detection of DOM-based XSS. In *20th ACM Conference on Computer and Communications Security Berlin 4.11.2013*.

[37] Zhuowei Li, XiaoFeng Wang, and Jong Choi. 2007. SpyShield: Preserving privacy from spy add-ons. In *Recent Advances in Intrusion Detection*. Springer, 296–316.

[38] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Toward Detecting and Preventing DOM Cross-Site Scripting. (2018).

[39] Mozilla. 2018. Modifying a web page. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Modify_a_web_page.

[40] Mozilla. 2018. SpiderMonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[41] PC Magazine. 2018. 'Web Of Trust' Browser Extension Cannot Be Trusted. http://www.pcmag.com/news/349328/web-of-trust-browser-extension-cannot-be-trusted.

[42] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for JavaScript. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 513–528.

[43] Oleksii Starov and Nick Nikiforakis. 2017. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1481–1490.

[44] StatCounter. 2018. Browser Market Share Worldwide. http://gs.statcounter.com/.

[45] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise client-side protection against DOM-based Cross-Site Scripting. In *USENIX Security Symposium*.

[46] V8 Project. 2018. Launching Ignition and TurboFan. https://v8project.blogspot.com/2017/05/launching-ignition-and-turbofan.html.

[47] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis.. In *NDSS*, Vol. 2007. 12.

[48] 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. *Information Security* (2012), 188–203.

[49] WebProNews. 2018. The Chromium-Powered Opera Is Finally Here. https://www.webpronews.com/the-chromium-powered-opera-is-finally-here/.

[50] Zachary Weinberg, Eric Y Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. 2011. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 147–161.

[51] Michael Weissbacher. 2018. These Chrome extensions spy on 8 million users. http://mweissbacher.com/blog/2016/03/31/these-chrome-extensions-spy-on-8-million-users/.

[52] Michael Weissbacher, Enrico Mariconti, Guillermo Suarez-Tangil, Gianluca Stringhini, William Robertson, and Engin Kirda. 2017. Ex-Ray: Detection of History-Leaking Browser Extensions. In *Annual Computer Security Applications Conference*. ACM publishing.