Hybrid Security Analysis of Web JavaScript Code via Dynamic Partial Evaluation

Omer Tripp
IBM Watson Research Center
Yorktown Heights, NY, USA
otripp@us.ibm.com

Pietro Ferrara
IBM Watson Research Center
Yorktown Heights, NY, USA
pietroferrara@us.ibm.com

Marco Pistoia
IBM Watson Research Center
Yorktown Heights, NY, USA
pistoia@us.ibm.com

ABSTRACT

This paper addresses the problem of detecting JavaScript security vulnerabilities in the client side of Web applications. Such vulnerabilities are becoming a source of growing concern due to the rapid migration of server-side business logic to the client side, combined with new JavaScript-backed Web technologies, such as AJAX and HTML5. Detection of client-side vulnerabilities is challenging given the dynamic and event-driven nature of JavaScript. We present a hybrid form of JavaScript analysis, which augments static analysis with (semi-)concrete information by applying partial evaluation to JavaScript functions according to dynamic data recorded by the Web crawler. The dynamic component rewrites the program per the enclosing HTML environment, and the static component then explores all possible behaviors of the partially evaluated program (while treating user-controlled aspects of the environment conservatively).

We have implemented this hybrid architecture as the JSA analysis tool, which is integrated into the IBM AppScan Standard Edition product. We formalize the static analysis and prove useful properties over it. We also tested the system across a set of 170,000 Web pages, comparing it with purely static and dynamic alternatives. The results provide conclusive evidence in favor of our hybrid approach. Only 10% of the reports by JSA are false alarms compared to 63% of the alarms flagged by its purely static counterpart, while not a single true warning is lost. This represents a reduction of 94% in false alarms. Compared with a commercial testing algorithm, JSA detects vulnerabilities in $>$ 4x more Web sites with only 4 false alarms.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Correctness proofs

General Terms

Algorithms, Experimentation, Security, Verification

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07...$15.00
http://dx.doi.org/10.1145/2610384.2610385

Keywords

hybrid analysis, JavaScript, security, partial evaluation

1. INTRODUCTION

In recent years, there has been a clear trend of migrating business logic from the server side to the client side of Web applications [22]. Web 2.0, AJAX, HTML5 and Rich Internet Applications (RIAs) feature sophisticated client-side functionality backed by JavaScript functions embedded into the HTML page. At the same time, however, the threat caused by client-side JavaScript vulnerabilities is also becoming more concrete.

Background.

The Document Object Model (DOM) is a structural representation of HTML and XML documents. As such, it provides an interface for manipulating the document. More specifically, the DOM is structured as a tree, whose nodes are objects with properties and methods. The DOM representation integrates with scripting languages—primarily JavaScript—to perform modifications to the style and content of the document, validate user input, etc.

While dynamic DOM modifications are highly useful, forming the foundation of all recent Web technologies, there are also serious security implications. Client-side JavaScript can modify the DOM in unintended ways that may lead to security vulnerabilities. The two primary categories of client-side vulnerabilities are *DOM-based Cross-site Scripting* (XSS) and *open redirect*. Both result from unsafe access to JavaScript objects such as `document.URL`, `document.referrer` and `document.location` by client-side JavaScript, as illustrated below:

```
var pos=document.URL.indexOf('val=')+4;
var val=document.URL.substring(pos,document.URL.length);
document.write(val); // DOM-based XSS
document.location.href = val; // open redirect
```

The HTML page is vulnerable to DOM-based XSS if an attacker can inject HTML markup into the values of parameters rendered by the client-side JavaScript code (for the example above: `...?val=<script>alert(...)</script>`). Open redirect arises if the JavaScript code redirects to the parameter value without proper validation (for the example above: `...?val=www.evil.com`). In this paper, general references to *client-side vulnerabilities* should be construed as these two types of vulnerabilities.

Detection of client-side vulnerabilities is challenging for both static- and dynamic-analysis tools. The main compli-

cation is to build an effective model of the behavior of JavaScript functions. By nature, JavaScript is a dynamic and flexible programming language, which supports persistent side effects through the DOM. JavaScript code is also organized into abstraction layers by popular client-side frameworks, such as Dojo [1] and jQuery [2]. Finally, JavaScript code is often bound to its enclosing HTML context, referencing the URL, input fields, etc. for purposes such as validation, redirection, content retrieval and rendering.

Existing Approaches.

The dynamic nature of JavaScript favors hybrid-analysis approaches, as will be discussed more in detail in Section 6. *Blended taint analysis* [21] has recently been proposed as one such approach. In this solution, static taint analysis is applied to dynamic execution traces of a JavaScript function, and the results are then combined into an overall security report. Other hybrid approaches have been proposed for run-time prevention of security attacks. These include dynamic data tainting, where a complementary static analysis accounts for control dependencies [20], as well as staging of information flow properties [6], where static analysis is applied to currently known JavaScript code, and a set of residual checks is performed on the remaining code when it is dynamically loaded.

A main challenge that remains, and we focus on, is how to perform security analysis of JavaScript code, such that dynamic features are accounted for, and at the same time, the analysis achieves high coverage of code behaviors. Blended analysis is restricted to a finite number of execution traces, and thus a narrow view of the entire program. Prevention techniques incur non-negligible run-time overhead, which is problematic for browser vendors, who are in constant competition over the performance of JavaScript execution.

Our Approach.

We propose a practical approach for security analysis of client-side JavaScript. The key idea is to apply static analysis to JavaScript content retrieved and rendered by a Web crawler, where dynamic information available through the crawler is used to perform partial evaluation of the JavaScript code. Specifically, properties read from the `document` object—such as `location`, `referrer` and `URL`, as well as certain other DOM accesses (via `getElementById(...)`)—are substituted by partially concretized values.

Our approach is informed by the observation that JavaScript code typically assumes, and references, a particular context induced by the HTML page enclosing it. As an example, JavaScript functions are used to manipulate the URL of their enclosing HTML page, validate the values of DOM attributes before sending them to the server side, etc. Making the concrete values referenced by the JavaScript code available to the static analysis algorithm enables more precise analysis. However, to retain high coverage, concretization of DOM access expressions is done carefully, leaving user-controlled portions of the DOM value fully abstract (for example, the query-string portion of the URL). An underlying assumption is that the portions of the DOM used for partial evaluation remain fixed. Indeed, portions of the DOM that access user input, such as the URL, cannot be modified without redirecting to another page, which shows this assumption to be viable.

There are a number of advantages to this approach: First, the integration between the static and dynamic components is loose. The static/dynamic interface is lightweight, consisting (only) of provision of dynamic values to the static analysis, where other hybrid approaches demand more complex integrations (for example, trace recording, and thus JavaScript instrumentation, for blended analysis). Second, the static analysis accounts for all program behaviors modulo concretization of expressions that are beyond the attacker's control, such as the host-path portion of the URL and the referrer. Our experiments over 170,000 distinct HTML pages indicate that there is no noticeable loss of coverage due to this restriction. Finally, replacing DOM access expressions with concrete values permits the analysis to track string values, which is strictly more precise than taint analysis. We describe a lightweight form of string analysis that we have designed and implemented within this hybrid setting.

To summarize, these are our principal contributions:

Novel hybrid analysis: We present a novel approach for security analysis of client-side JavaScript. In our approach, static string analysis is applied to a partially evaluated version of the JavaScript code in which certain DOM expressions that are frequently accessed by JavaScript functions, are security relevant, and concretized per the dynamic context. To maximize coverage, concretization is done in a controlled way, to ensure that user-provided portions of the DOM expression are treated conservatively.

Formal description and proofs: We have formalized our approach as an instance of abstract interpretation. We provide a rigorous description of our static analysis, backed by motivations for the design choices governing it, and prove that the analysis is (i) distributive (and can thus be computed precisely and efficiently), (ii) sound (that is, it overapproximates all the possible runtime behaviors), and (iii) guaranteed to terminate.

Implementation and evaluation: We have implemented our analysis as the JSA tool, which we recently integrated into IBM Security AppScan Standard Edition, a commercial product for black-box Web security testing in which JSA complements the server-side analysis engine by performing client-side security assessment. Integrating JSA into a dynamic product was enabled thanks to its low rate of false alarms, which we measured across a set of over 170,000 Web pages. We include a comparison with purely static and dynamic alternatives. The results are conclusive in favor of JSA: Only 10% of the reports by JSA are false alarms compared to 63% of the alarms flagged by a commercial-grade static taint analysis, while not a single true issue is lost. This represents a reduction of 94% in false alarms. Compared with a commercial testing algorithm, JSA is able to detect vulnerabilities in $>$ 4x more Web sites with only 4 false alarms.

2. MOTIVATION

JSA was designed to meet three primary requirements: accuracy, coverage and performance. Beyond the general tension between these requirements, there are also challenges that are specific to JavaScript. For coverage, JavaScript is event driven (with subtle dependencies between events), which favors static analysis [11]. For accuracy, JavaScript functions are typically rich in string manipulations, which motivates string rather than taint analysis. Finally, for performance, the analysis must be lightweight, especially given that today's Web sites consist of thousands of Web pages, which often each contain hundreds of JavaScript functions.

```
1 var search_term = 'login.html';
2 var str = document.URL; // source
3 var url_check = str.indexOf(search_term);
4 if (url_check > -1) {
5   var result = str.substring(0, url_check);
6   result = result + 'login.jsp' + str.substring(
7                     (url_check + search_term.length), str.length);
8   document.URL = result; } // sink
```

Figure 1: Real-world JavaScript code that performs safe redirection

2.1 Illustrative Example

The JavaScript code in Figure 1, extracted from a Web page in the Alcatel-Lucent Web site under URL `https://market.alcatel-lucent.com/release/jsp/sso/login.html`, (conditionally) performs redirection via assignment to the `URL` field of the `document` object. Standard analysis techniques based on information-flow (or taint) tracking [10, 13, 18, 19] would flag this code as vulnerable to open-redirect attacks, since there is a data-flow path from the statement reading the URL (marked as `source`) to the statement performing the redirection (marked as `sink`). This flow is potentially problematic because the user is able to set portions of the URL string (in particular, parameter values and hash marks), and so the redirection target may be influenced by user-controlled values.

Careful review of the code in Figure 1 suggests, however, that the user is unable to affect the redirection operation, and thus taint analysis would raise a false alarm. This is because the code merely replaces `'login.html'` with `'login.jsp'` via calls to `indexOf` (to locate `login.html`), `substring` (to obtain the prefix up to `'login.html'`) and then the `+` (concatenation) operator (to append the URL suffix to `'login.jsp'`). Importantly, the entire URL prefix up to the first occurrence of `'login.html'`, as well as the suffix beyond `'login.html'`, are preserved when the redirection target is constructed. This implies, in specific, that the user cannot affect the host path.

This example is not only real, but also representative of a wide class of JavaScript functions whose purpose is to perform controlled edits to the URL string and redirect to the resulting URL. (For more examples, see Figure 7.) Within this class, there is the notable (trivial) case of self redirection (i.e., `document.URL = document.URL`), where the effect is to refresh the page following DOM modifications by JavaScript code. Similar sources of noise arise for DOM-based XSS. A common example is JavaScript code that sets event handlers, where a vulnerability manifests only if the attacker has full control over the assigned string value, and can force it to begin e.g. with `'javascript:'`. This is often not the case, resulting in a false alarm for taint-based techniques.

2.2 Main Idea

The important observation informing our approach is that JavaScript code typically refers to the environment in which it is run. The code in Figure 1, for instance, refers to the concrete strings `'login.html'` and `'login.jsp'`, where `'login.html'` is indeed a substring of the URL that serves that code. This observation leads to a new form of hybrid analysis, whereby the dynamic component makes concrete DOM content available to the static analysis. The static component, in turn, can rewrite expressions such as `document.location.href` as partially concrete values, where user-controlled subvalues are treated conservatively (e.g., the URL host path versus the query string).

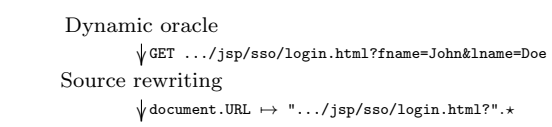

Figure 2: Hybrid flow enforced by the JSA system

Figure 2 illustrates the resulting hybrid flow. The dynamic oracle collects relevant DOM information and passes it to a rewriting module, which links references to the DOM within JavaScript code with partially concretized values (e.g., read access to the URL is replaced by a partially concrete string consisting of a concrete host path and an "abstract" query string represented as the $.\star$ regular expression).

This form of static/dynamic cooperation, translating into partial evaluation of the JavaScript code per the concrete environment it references, addresses the requirements above:
Accuracy: Availability of (semi-)concrete values enables new forms of string analysis, rather than the traditional approach of taint analysis. For the code in Figure 1, for instance, the analysis can conclude that the host path is beyond user control, as JSA indeed does, by modeling the string operations. The analysis can also leverage string information for better classification of findings. As an example, for a sink statement *snk* that renders content to the DOM (e.g., via assignment to the `innerHTML` field of an `Element` object), an unchecked string value *s* flowing into *snk* would lead to different types of vulnerabilities depending on whether *s* starts with `javascript:` or `mailto:`. In the former case, the attacker can execute a script on the victim's browser, whereas in the latter case the attacker can control the address to which email content will be sent.
Coverage: While static analysis is applied to a specialized version of the JavaScript code, where DOM references are replaced by (semi-)concrete values, specialization is done carefully (abstracting away user inputs) and the analysis still considers all feasible execution paths of the specialized code (including looping, conditional branching, etc). This offers greater coverage than purely dynamic approaches or techniques that apply static analysis to a finite number of dynamic traces [21].
Performance: For both implementability and performance, the analysis is designed such that the static and dynamic components are loosely coupled, and share a minimal interface of data exchange. Further, string analysis can be optimized by staging it, the first step being a standard taint analysis. We have implemented this optimization in JSA.

3. TECHNICAL OVERVIEW

In this section, we provide a detailed technical overview of the static analysis algorithm.

3.1 Staged Algorithm

At the technical level, our system benefits from the concrete data made available by the Web crawler by perfoming

Input: JavaScript call graph G
Input: Dynamic oracle $\mathcal{O}$
Output: Security vulnerabilities $V = \{v_1, \ldots, v_n\}$

```
1  begin
2      Src ⟵ scan G for source statements
3      Snk ⟵ scan G for sink statements
4      V ⟵ ∅
5      foreach (st, st') ∈ Src × Snk do
6          r ⟵ compute data-flow reachability from st to st'
7          if r then
8              e' ⟵
               partially evaluate RHS e of st by querying O
9              e# ⟵ abstract e' as a string prefix
10             π ⟵
               perform string analysis with (st, e#) as seed
11             {e#_1, ..., e#_k} ⟵ π[st']   /* query π at loc st'
               */
12
13             for 1 ≤ i ≤ k do
14                 if e#_i flowing into st' is unsafe then
15                     V ⟵
                       V ∪ [make report for st e# ⇝ e#_i st']
16                 end
17             end
18         end
19     end
20     return V
21 end
```

Algorithm 1: The core JSA algorithm

a lightweight form of string analysis as part of a staged analysis algorithm whose first step is standard taint tracking. The complete algorithm is outlined in Algorithm 1. We simplify the actual algorithm, omitting several obvious optimizations, for brevity and readability. The inputs are (i) a call graph over the JavaScript functions in the HTML page assembled by the Web crawler, as well as (ii) a dynamic oracle mediating access to concrete DOM values. The output is a set of security vulnerabilities detected over the call graph.

The first step (lines 2–3) is to scan the call graph for sources and sinks. For the code in Figure 1, these are the read and write accesses to `document.URL`. Unlike the case of typed programming languages (like Java), where locating sources and sinks merely requires syntactic matching, the dynamic semantics of JavaScript mandates full-fledged data-flow analysis, which we perform similarly to Guarnieri et al. [13]. As an example, a popular way of rendering content to the DOM is to invoke the `document.getElementById(...)` function and then write to the `innerHTML` field of the returned object. Assignment to `innerHTML` is thus a DOM-based-XSS sink. To detect instances of this sink, the analysis must first compute all the aliases of the `document` object (via pointer analysis), then find all calls to `getElementById(...)` and record the returned objects, and finally locate assignments to the `innerHTML` field on these objects.

Next, for each source/sink pair (st, st'), the algorithm performs staged analysis to optimize performance. The first phase (line 6) is to check if there exists a (sanitizer-agnostic) data-flow path between the source and the sink. If data from the source does not reach the sink, then the source/sink pair at hand cannot lead to any vulnerability, and no further analysis is required.

Otherwise, if data from the source flows into the sink (as indeed occurs in Figure 1, where data from `str` flows into the assignment to `document.URL` through the `substring` and `+` operations), the right-hand-side (RHS) expression of the source statement is partially evaluated (line 8), resulting in an abstract string value consisting of fully concrete alongside fully abstract segments (line 9; `"https://market.alcatel-lucent.com/release/jsp/sso/login.html?".`$\star$). This string value is propagated forward until a fixpoint is reached (line 10). The fixpoint solution π is then queried at the sink location (line 11). For any abstract value $s_i^{\#}$ flowing into the sink that may concretize into a security payload (line 14), a report is created (line 15). The set of all reports is returned after all source/sink pairs have been inspected (line 20).

3.2 String Analysis

The novel step in Algorithm 1 is the string analysis computed as a refinement of the taint analysis. String analysis is enabled thanks to the concrete DOM values provided by the dynamic component. Section 4 describes the analysis at the formal level as an abstract-interpretation problem. Our purpose here, instead, is to provide a higher-level overview with motivations for our design decisions.

String abstraction.

As observed earlier, JavaScript code typically assumes, and references, a particular DOM context being an integral part of the HTML. Also, because the DOM is amenable to direct access via the `window` and `document` properties, as well as the `getElementById(...)` function, the retrieved values are often not persisted into the heap. These unique characteristics of JavaScript coding have led us to a string abstraction that represents user-agnostic values (or subvalues) concretely. Specifically, our string abstraction consists of a concrete prefix and a possibly unknown suffix, which captures the structure of all DOM values that our analysis deems relevant. For instance, this abstraction tracks very precise information on the URLs produced by the example in Figure 1 considering the (semi-)concretized value produced by the rewriting module of our system.

Our analysis tracks string values through local variables and procedure calls, but models the heap coarsely as a single abstract element, such that flow through the heap is represented cheaply albeit imprecisely. This is in line with our empirical observation that DOM values of interest often do not escape into the runtime heap. Flow through local variables and calls is handled by tracking environment pointers to strings. We also track integral values that result from `indexOf` queries to account precisely for `substring` operations.

Distributive analysis.

Local pointers to strings, as well as integral offsets, are folded onto the string abstraction. This design choice lets us formulate the analysis as a distributive data-flow problem, which can be solved precisely in polynomial time as an instance of the IFDS framework [16]. Representing pointers to a string as part of its abstraction satisfies the distributivity requirement over the confluence operator. An abstract transformer is the natural lifting of point-wise microtransformers, which operate on individual data-flow facts. This is enabled because the data-flow fact refers not only to the abstract string value but also to the pointers and integers required to model string operations accurately.

```
1 var strUrl = document.location.href;
2 var lwrStrUrl = strUrl.toLowerCase();
3 var n = lwrStrUrl.indexOf("bc-");
4 if (n != -1)
5   document.location.href = strUrl.substring(0, n);
```

Figure 3: Real-world example demonstrating the common pattern of querying a lower-/upper-case version of a string, then applying a transformation to the original string

$\mathtt{x} := \mathtt{y}.\mathtt{substring}(\mathtt{n}_1, \mathtt{n}_2)$
$\mathtt{x} := \mathtt{y}.\mathtt{toLowerCase}()$
$\mathtt{x} := \text{"}\mathtt{str}\text{"}[.*]$
$\mathtt{x} := \mathtt{y} \circ \mathtt{z}$
$\mathtt{n} := \mathtt{x}.\mathtt{indexOf}(\mathtt{c})$
$\mathtt{n} := \mathtt{x}.\mathtt{length}()$

Figure 4: String operations

$\mathsf{Prx} = \mathsf{String} \times \{\mathsf{tt}, \mathsf{ff}\}$
$\mathsf{Idx} = \mathsf{IntVar} \to \mathbb{Z} \cup \{\top_{\mathbb{Z}}\}$
$\mathsf{PPrx} = \wp(\mathsf{StrVar} \times \mathsf{Prx}) \times \mathsf{Idx}$
$\Sigma = \wp(\mathsf{PPrx})$

Figure 5: Abstract domain

Useful correlations.

As a final technical point, we discuss our choice to break the abstract state into atomic data-flow facts that represent partitions (potentially consisting of more than one string abstraction) rather than individual strings. This design choice follows from the prevalence, in the wild, of examples like the one in Figure 3. In this example, taken from an internal IBM webpage whose host path ends in `ETWeb/Help/BC-approving...cards.html`, the location string pointed-to by `strUrl` is converted to lower case as `lwrStrUrl`. `lwrStrUrl` is then searched for the index of `"bc-"` as `n`, and then — if this substring is found — `n` flows into the `substring` operation applied to `strUrl`. This pattern is prevalent because URLs are case insensitive, but string searching is case sensitive.

The important challenge is to correlate between the `indexOf` operation computed on `lwrStrUrl` and the `substring` operation computed on `strUrl`, which are dependent via the index `n`. The distributive setting prohibits sharing of information between data-flow facts. This naturally leads to the idea of accommodating correlated string values into the same data-flow fact, which then becomes a partition.

4. FORMAL DESCRIPTION

In this Section, we formalize the abstract domain that approximates information about strings. In the implementation, we first run a constant-propagation analysis to propagate constant strings and integer values. In the formalization, we omit this part, since it represents a well-known analysis and it is not a contribution of our work. Nevertheless, we implemented this pre-processing step in our tool to enhance the precision of the analysis in practice.

4.1 Notation

Let String be the set of all finite strings. Given a string $\mathsf{s} \in \mathsf{String}$ and a character c,

- $lwr(\mathsf{s})$ represents the transformation of s into lower case characters;
- $\mathsf{c} \in \mathsf{s}$ denotes that c appears in s at least once;
- $firstIndexOf(\mathsf{s}, \mathsf{c})$ returns the index of the first occurrence of c in s or -1 if $\mathsf{c} \notin \mathsf{s}$;
- $\mathsf{s}_1 \circ \mathsf{s}_2$ concatenates the two strings;
- $sbstr(\mathsf{s}, \mathsf{v}_1, \mathsf{v}_2)$ returns the substring of s from the v_1-th to the v_2-th character; and
- $lng(\mathsf{s})$ returns the number of characters of s.

4.2 String Operations

We focus our formalization on the core set of string operations listed in Figure 4, where "`str`".$*$ represents a string constant `str` followed by an unknown suffix. For brevity, and to avoid repetition, we restrict our formal definitions to these standard representative operations, but emphasize that our implementation supports all the standard string operations of the JavaScript language. Most of the remaining operations (e.g., other overloads of `substring`(...), `toUpperCase`(), etc) can be formalized straightforwardly based on this core set. We assume the ability to distinguish between string and integer variables within the context of string operations. Therefore, given set Var of local variables, we suppose that it is partitioned into integer and string variables (IntVar and StrVar, respectively). Formally, $\mathsf{Var} = \mathsf{IntVar} \cup \mathsf{StrVar}$. In Figure 4, `x`, `y`, and `z` belong to StrVar, whereas `n`, $\mathtt{n}_1$, and $\mathtt{n}_2$ belong to IntVar. We define by St the set of all program statements as defined in Figure 4.

4.3 Abstract Domain

Figure 5 formalizes our abstract domain Σ. Let Prx be the set of prefixes. An abstract value consists of (i) a string representing the prefix and (ii) a boolean flag denoting whether or not the prefix may have a suffix. Idx tracks numerical information on variables representing indexes into strings. This is represented by a function relating each numerical variable to either its constant value or $\top_{\mathbb{Z}}$ (representing any possible numerical value). Then, a partition in PPrx collects the abstract prefixes of some string variables sharing the same prefix (modulo case sensitivity) as well as the values of integer variables representing indexes into these strings. Finally, the abstract domain Σ collects a set of partitions in PPrx. The lattice structure is based on set operators. Formally, $\langle \Sigma, \cup, \cap, \mathsf{PPrx}, \emptyset \rangle$.

4.3.1 Properties of the Abstract Domain

Elements in PPrx are intended to partition the string variables into strings having the same (case-insensitive) prefix, as motivated in Section 3.2. Therefore, we need to restrict the elements belonging to our abstract domain Σ. First of all, given an element $(\mathsf{P}, \mathsf{i}) \in \mathsf{PPrx}$, all the prefixes in the partition are the same up to case sensitivity. To compare different prefixes, we utilize their lower-case representation.

PROPERTY 1 (SAME PREFIX).

$$\forall (\mathsf{P}, \mathsf{i}) \in \mathsf{PPrx}.\ \forall (\mathtt{x}_1, (\mathtt{p}_1, \mathtt{l}_1)).\ (\mathtt{x}_2, (\mathtt{p}_2, \mathtt{l}_2)) \in \mathsf{P} : lwr(\mathtt{p}_1) = lwr(\mathtt{p}_2)$$

Then, given an element $(\mathsf{P}, \mathsf{i}) \in \mathsf{PPrx}$ where $\mathsf{P} \neq \emptyset$, its canonical representation is the lower-case representation of its prefix. Thanks to Property 1, we know that all the elements in P agree on this prefix.

DEFINITION 1 (CANONICAL REPRESENTATION). *The canonical representation of an element in* Prx *is defined as the lower-case representation of the prefixes of its elements. Formally,* $can(\mathsf{P}) = lwr(\mathtt{p}) : \exists (\mathtt{x}, (\mathtt{p}, \mathtt{l})) \in \mathsf{P}$.

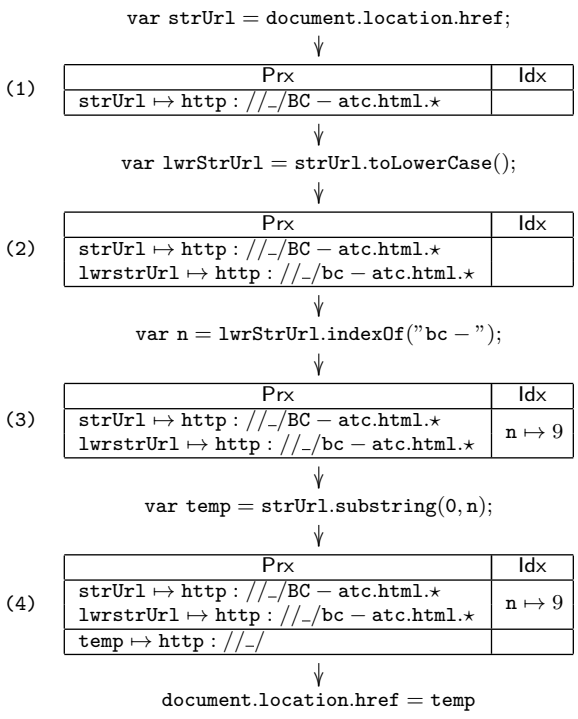

Figure 6: The results of our analysis on the running example of Figure 3 (To reduce space, we abbreviate the concrete URL as _/BC-atc.html?.⋆)

4.4 Abstract Semantics

We now define the abstract semantics of the statements in Figure 4 on the abstract domain formalized in Section 4.3.

First of all, we define support functions to forget what we know in a specific partition about integer and string variables. Formally,

$$fgtStrVar(\mathtt{x}, \mathsf{P}) = \{(\mathtt{y}, \mathtt{s}) : (\mathtt{y}, \mathtt{s}) \in \mathsf{P} \wedge \mathtt{y} \neq \mathtt{x}\}$$
$$fgtIdx(\mathtt{n}, \mathsf{i}) = [\mathtt{n}' \mapsto \mathsf{i}(\mathtt{n}') : \mathtt{n}' \in dom(\mathsf{i}) \wedge \mathtt{n}' \neq \mathtt{n}]$$

We will use these functions to forget what we knew about the assigned variable before the statement.

We now define the semantics of a statement $\mathbb{S}_{\mathsf{PPrx}} : \mathsf{PPrx} \rightarrow \wp(\mathsf{PPrx})$ that, given a specific partition, returns the partitions approximating the exit state starting from the given entry partition. For the sake of simplicity, we focus our formal definitions on the partitions containing some information on the right-hand side of the assignment. For the other partitions, we simply forget the assigned string or integer variables by applying *fgtStrVar* and *fgtIdx*, respectively.

The semantics of $\mathtt{x} :=$ "str" propagates the information contained in the entry partition, and additionally adds a partition relating x to the prefix str without a suffix. Formally,

$$\mathbb{S}_{\mathsf{PPrx}}[\![\mathtt{x} := \text{"}\mathtt{str}\text{"}, (\mathsf{P}, \mathsf{i})]\!] = \{(\{(\mathtt{x}, (\mathtt{str}, \mathsf{ff}))\}, \emptyset)\}$$
$$\mathbb{S}_{\mathsf{PPrx}}[\![\mathtt{x} := \text{"}\mathtt{str}\text{"}.\star, (\mathsf{P}, \mathsf{i})]\!] = \{(\{(\mathtt{x}, (\mathtt{str}, \mathtt{tt}))\}, \emptyset)\}$$

Running example: The abstract semantics of the assignment at line 1 of our running example relates variable strURL to the prefix http : //_/BC − atc.html.⋆. This is represented by state (1) in Figure 6. □

The semantics of $\mathtt{x} :=$ y.toLowerCase() augments the information tracked by the given partition if it contains y, and in particular it relates x to the lower case representation of the prefix of y. Formally,

$$\mathbb{S}_{\mathsf{PPrx}}[\![\mathtt{x} := \mathtt{y}.\mathtt{toLowerCase}(), (\mathsf{P}, \mathsf{i})]\!] = \\ = \{(\mathsf{P}' \cup \{(\mathtt{x}, (lwr(\mathsf{p}), \mathsf{b}))\}, \mathsf{i}) : (\mathtt{y}, (\mathsf{p}, \mathsf{b})) \in \mathsf{P})\}$$

where $\mathsf{P}' = fgtStrVar(\mathtt{x}, \mathsf{P})$.

Running example: The abstract semantics of the assignment at line 2 adds variable lwrStrUrl to state (1) in Figure 6, and relates it to http : //_/bc − atc.html.⋆. Therefore, we obtain the state (2) in Figure 6. □

The semantics of $\mathtt{n} :=$ x.indexOf(c) affects the Idx component. In particular, if in the given partition we can determine the exact index of the given character in the given string, then we store this numerical value. Instead, if the character is not in the prefix and there is no suffix, then we are certain that the string does not contain the given character, and therefore we know that indexOf returns -1. Otherwise, we cannot infer a precise numerical value for the index, and thus we conservatively store $\top_{\mathbb{Z}}$. Formally,

$$\mathbb{S}_{\mathsf{PPrx}}[\![\mathtt{n} := \mathtt{x}.\mathtt{indexOf}(\mathtt{c}), (\mathsf{P}, \mathsf{i})]\!] = \{(\mathsf{P}, \mathsf{i}'') : (\mathtt{x}, (\mathsf{p}, \mathsf{b})) \in \mathsf{P} \wedge \\ \mathsf{i}'' = \mathsf{i}'\left[\mathtt{n} \mapsto \begin{cases} firstIndexOf(\mathsf{p}, \mathtt{c}) & \text{if } \mathtt{c} \in \mathsf{p} \\ -1 & \text{if } \mathtt{c} \notin \mathsf{p} \wedge \mathsf{b} = \mathsf{ff} \\ \top_{\mathbb{Z}} & \text{if } \mathtt{c} \notin \mathsf{p} \wedge \mathsf{b} = \mathtt{tt} \end{cases}\right]\}$$

where $\mathsf{i}' = fgtIdx(\mathtt{n}, \mathsf{i})$.

Running example: The semantics augments the Idx component of state (2), relating n to the index pointing to "bc − " in lwrStrUrl (that is, to the 9th character). This is represented by state (3) in Figure 6. □

Similarly, the semantics of $\mathtt{n} :=$ x.length() affects only the Idx component. In particular, if we can determine the exact length of x — that is, if we have x in the current partition, and it does not contain a suffix — then we store it. Alternatively, if x may contain a suffix, then we conservatively store $\top_{\mathbb{Z}}$. Formally,

$$\mathbb{S}_{\mathsf{PPrx}}[\![\mathtt{n} := \mathtt{x}.\mathtt{length}(), (\mathsf{P}, \mathsf{i})]\!] = \{(\mathsf{P}, \mathsf{i}'') : (\mathtt{x}, (\mathsf{p}, \mathsf{b})) \in \mathsf{P} \wedge \\ \mathsf{i}'' = \mathsf{i}'\left[\mathtt{n} \mapsto \begin{cases} lng(p) & \text{if } \mathsf{b} = \mathsf{ff} \\ \top_{\mathbb{Z}} & \text{if } \mathsf{b} = \mathtt{tt} \end{cases}\right]\}$$

where $\mathsf{i}' = fgtIdx(\mathtt{n}, \mathsf{i})$.

The semantics of $\mathtt{x} := \mathtt{y} \circ \mathtt{z}$ assigns, if the given partition tracks information on y, the prefix of y to x, and assumes that an arbitrary suffix could follow. This remains the case even if information about z may be tracked by another partition. A sound and distributive analysis is prohibited from observing that information in general, and so even if we know that the prefix y is not followed by a suffix, we must account for the effect of the concatenation conservatively. Formally,

$$\mathbb{S}_{\mathsf{PPrx}}[\![\mathtt{x} := \mathtt{y} \circ \mathtt{z}, (\mathsf{P}, \mathsf{i})]\!] = \\ \{(\mathsf{P}'', \mathsf{i}) : \mathsf{p}'' = \mathsf{P}' \cup \{(\mathtt{x}, (\mathsf{p}, \mathtt{tt}))\} \wedge (\mathtt{y}, (\mathsf{p}, \mathsf{b})) \in \mathsf{P}\}$$

where $\mathsf{P}' = fgtStrVar(\mathtt{x}, \mathsf{P})$.

Finally, the semantics of $\mathtt{x} :=$ y.substring($\mathtt{n}_1, \mathtt{n}_2$) has to take into account various cases if the given partition tracks the prefix of y. In particular, if the prefix of y is longer than $\mathsf{i}(\mathtt{n}_2)$ characters, then we have precise information on the computed substring. Instead, if it is longer than $\mathsf{i}(\mathtt{n}_1)$ characters but shorter than $\mathsf{i}(\mathtt{n}_2)$, then we have precise information about the prefix, though it could be followed by an

unknown suffix. Finally, if it is shorter than $\mathsf{i}(\mathtt{n}_1)$ characters, then we cannot infer any information on `x`. This intuition is formalized as follows.

First of all, we introduce the *cut* function that — given an abstract state, a local variable, as well as begin and end index variables — returns the abstract prefix in the domain, which represents the partition that should represent this substring. This is formalized as follows assuming that indexes are mapped to concrete integer values:

$$cut(\mathsf{P},\mathsf{i},\mathtt{x},\mathtt{n}_1,\mathtt{n}_2) = \{\mathsf{r} : \exists(\mathtt{x},(\mathsf{p},\mathsf{b})) \in \mathsf{P}, \mathsf{v}_1 = \mathsf{i}(\mathtt{n}_1), \mathsf{v}_2 = \mathsf{i}(\mathtt{n}_2)$$
$$\mathsf{r} = \begin{cases} (sbstr(\mathsf{p},\mathsf{v}_1,\mathsf{v}_2),\mathsf{ff}) & \text{if } lng(\mathsf{p}) \geq \mathsf{v}_2 \\ (sbstr(\mathsf{p},\mathsf{v}_1,lng(\mathsf{p})),\mathsf{tt}) & \text{if } lng(\mathsf{p}) < \mathsf{v}_2 \wedge \\ & lng(\mathsf{p}) \geq \mathsf{v}_1 \wedge \mathsf{b} = \mathsf{tt} \end{cases} \}$$

In the remaining cases, where index values are missing, we take a conservative approach and refrain from computing a concrete string value, since we cannot establish any prefix for the resulting string.

Using this function, the abstract semantics updates the partitions as follows:

$$\mathbb{S}_{\mathsf{PPrx}}[\![\mathtt{x} := \mathtt{y}.\mathtt{substring}(\mathtt{n}_1,\mathtt{n}_2),(\mathsf{P},\mathsf{i})]\!] =$$
$$= \{(\{(\mathtt{x},(\mathsf{p},\mathsf{b}))\},\emptyset) : (\mathsf{p},\mathsf{b}) \in cut(\mathsf{SP},\mathtt{y},\mathtt{n}_1,\mathtt{n}_2)\}$$

Running example: cut returns $\mathtt{http} : //\_/$ when analyzing the statement at line 4 of our running example in state `(3)`. Then our abstract semantics adds the partition tracking that the prefix of `temp` is $\mathtt{http} : //\_/$ to the abstract state. This is represented by the last row of state `(4)` in Figure 6. □

The abstract semantics $\mathbb{S} : \mathbf{\Sigma} \to \mathbf{\Sigma}$ is the pointwise application of $\mathbb{S}_{\mathsf{PPrx}}$ to all the partitions contained in an abstract state. Formally,

$$\mathbb{S}[\![\mathtt{st},\mathsf{SP}]\!] = \bigcup_{(\mathsf{P},\mathsf{i})\in\mathsf{SP}} \mathbb{S}_{\mathsf{PPrx}}[\![\mathtt{st},(\mathsf{P},\mathsf{i})]\!]$$

THEOREM 1 (DISTRIBUTIVITY OF THE ABS. SEMANTICS). $\forall \mathsf{SP}_1,\mathsf{SP}_2 \in \mathbf{\Sigma}, \forall \mathtt{st} \in \mathsf{St} : \mathbb{S}[\![\mathtt{st},\mathsf{SP}_1 \cup \mathsf{SP}_2]\!] = \mathbb{S}[\![\mathtt{st},\mathsf{SP}_1]\!] \cup \mathbb{S}[\![\mathtt{st},\mathsf{SP}_2]\!]$

PROOF. *The proof follows straightforwardly from the definition of the abstract semantics, since this is defined as the union of the semantics on the single partitions, and therefore it is distributive w.r.t. the least upper bound of the abstract domains (that is, the set union operator).* □

The height of the lattice of our abstract domain may be infinite. Therefore, we enforce its finiteness by allowing prefixes of up to k characters in the domain. For instance, if this limit is set to 5, and we analyze `s` := "`abcdef`", then we would track the abstract prefix (`abcde`, `tt`) as an approximation of `s`. In practice, we set an extremely high bound of 10000 characters in our implementation (and experiments), that was never reached in practice.

THEOREM 2 (CONVERGENCE OF THE ANALYSIS). *The analysis converges in finite time.*

PROOF. *A sufficient condition to guarantee the convergence of the analysis is the finiteness of the height of the abstract domain, that we achieved by enforcing an upper limit k. Therefore, our analysis converges in a finite time.* □

Finally, we prove the soundness of our analysis, thereby establishing that no vulnerabilities are missed with respect to the concrete DOM at hand.

THEOREM 3 (SOUNDNESS OF THE ANALYSIS). *The abstract domain is a sound approximation of the concrete domain, and the abstract semantics overapproximates the concrete semantics.*

PROOF. *Domain: In the theory of abstract interpretation [7, 8], the concrete and abstract domains are related through respective abstraction and concretization functions. These form together a Galois connection. A common way of establishing and proving this relationship is by defining an abstraction function that is a complete join morphism (that is, it preserves the join operator), and then defining the concretization function accordingly (see Theorem 7 in [9]). Informally, the abstraction function, given an abstract partition in* PPrx*, returns all the abstract states that relate (1) each string variable in the partition to an admissible string (that is, a string starting with the given prefix and followed by a suffix, if the suffix component of* Prx *is* tt*), and (2) each numerical variable in the partition to its respective constant value, or to a random numerical value if it is $\top_{\mathbb{Z}}$. Then the abstraction function of $\mathbf{\Sigma}$ applies the abstraction of* PPrx *pointwise to all the partitions in the abstract state. Therefore, this abstraction function preserves the join operator (that in our abstract domain is the set-union operator), and is a complete join morphism.*
Semantics: We sketch an informal proof that the abstract semantics overapproximates the standard concrete semantics. In particular, we reason about statement $\mathtt{x} := \mathtt{y} \circ \mathtt{z}$. First of all, removing all the information we were previously tracking on x *through $fgtIdx(\mathsf{P},\mathtt{x})$ is sound w.r.t. the standard concrete semantics of assignment, since this removes the value pointed-to by* x *from the concrete environments. Then, if we have a partition containing* y*, we track that* x *after the assignment has the same prefix followed by an unknown suffix. This is sound since when we concatenate two strings in the concrete semantics, the resulting string indeed starts with the left-hand string of the concatenation, and therefore has the same prefix. Setting the suffix to* tt *guarantees soundness, since in this way we suppose that the right-hand string may be any string even if the left-hand side of the concatenation represents a constant string.* □

5. IMPLEMENTATION AND EVALUATION

This section describes (i) the implementation and productization of the JSA system, as well as (ii) two sets of experiments comparing JSA with static and dynamic alternatives.

5.1 Product Integration

We have implemented JSA atop the open-source WALA framework [3], which contains algorithms for JavaScript call-graph construction as well as an optimized implementation of the IFDS framework [16]. JSA is currently integrated into IBM Security AppScan Standard Edition, a commercial black-box security assessment product that performs testing of the client side as well as server side of Web applications. At a high level, AppScan consists of a Web crawler and a testing engine. The crawler builds a *site model*, which exposes the business logic and structure of the subject website. The testing engine then applies testing to the extracted Web pages and HTTP input points. JSA complements this core functionality by contributing warnings on client-side vulnerabilities into the main AppScan report. JSA utilizes the built-in AppScan crawler to gain access to HTML pages

with JavaScript content, where the crawler serves as the oracle providing dynamic information for partial evaluation.

An important feature of JSA, mentioned in Section 2.2, is fine-grained issue categorization. While an information-flow-based static analysis algorithm would only be able to distinguish DOM-based XSS from open redirect (based on the source/sink pair involved), JSA supports subcategories of these two broad categories. In our current implementation, under DOM-based XSS, JSA distinguishes between two variants: *direct*, where the attacker has full control over the rendered content; and *body*, where the attacker is restricted to the target of a `javascript:` handler. Under open redirect, JSA distinguishes between three variants: *absolute*, where the attacker can redirect to any chosen URL; *relative*, where the attacker can only navigate within a designated website; and *mailto*, where the attacker can influence the target of email sending.

5.2 Experimental Evaluation

We conducted two sets of experiments, where the objective was to assess the efficacy of JSA relative to two baseline configurations: (i) AppScan combined with a taint-analysis engine for static client-side security assessment (Section 5.2.2) and (ii) AppScan without JSA, relying solely on its dynamic client-side testing capabilities (Section 5.2.3). The numbers we obtained provide a clear indication that the JSA configuration is superior to both of these alternatives, avoiding almost all of the (many) false reports that configuration (i) suffers from while yielding substantially more true findings than configuration (ii).

5.2.1 Benchmark Suite

For our experimental study, we considered a collection of 675 real-world websites, including all Fortune 500 companies, the top 100 websites [4], as well as several handpicked sites of IT and security vendors. On each of these websites, the AppScan crawler performed passive crawling (i.e., without login and HTML form submissions) — to prevent from intrusive, or otherwise nonstandard, interaction with the site — with a limit of up to 500 webpages. This yielded about 250 webpages per site on average (where the limit of 500 pages was rarely reached) for a total of over 170,000 distinct webpages.

The running examples in Figures 1 and 3 are taken from our suite. Beyond these examples, we show in Figure 7 several more representative JavaScript codes to give the reader a better taste of the challenges that real-world JavaScript code poses to a security analysis algorithm. None of these examples contains vulnerabilities (we deliberately excluded examples that are vulnerable because we are also disclosing the respective website URLs), but coming to this conclusion (even manually) requires nontrivial analysis, as the reader can verify.

The example in Listing 1 is potentially vulnerable, although in reality all `changeZipRedirect(...)` invocations are with a constant `zipCodeRedirect` value (as suggested by the comment). This discourages both open-redirect and DOM-based XSS attacks even though `redirectStr` is user controlled. The example in Listing 2 redirects to a URL that contains portions of the current URL, but these are taken from the host-path string. This example is also immune to DOM-based XSS attacks, because the `javascript:` directive has the fixed behavior of performing redirection. Similarly, in Listing 3 `cId` and `pageUrl` both flow into the target URL's query string (i.e., after `?`). Finally, the example in Listing 4 computes a prefix of the URL string that lies within the host path, and then assembles the complete URL by appending the constant string `"livehelp/?pluginID="` followed by user input. This is, again, a safe form of redirection. To summarize, none of these three examples contains neither DOM-based-XSS nor open-redirect vulnerabilities, though in all cases there is source-to-sink data flow that is not mediated by any form of validation or sanitization.

5.2.2 Comparison with Taint Analysis

To compare JSA with a hybrid architecture featuring information -flow-based static client-side scanning, without the JSA notion of partial evaluation, we disabled the second step of the staged JSA algorithm while boosting the accuracy of the baseline algorithm by accounting for sanitizer methods. (See Algorithm 1.) We stress that the first step of the algorithm is not merely a naive implementation of JavaScript taint analysis, but a carefully designed and highly optimized analysis algorithm that was developed to meet industry standards before being replaced by JSA.

Methodology.

To compare between the engines, we ran both on the entire collection of webpages. The client-side vulnerability reports output by both engine configurations were then manually reviewed by a security expert from the IBM Application Security Research Group, who classified each report as either being a true positive or a false positive, where a true positive is a source/sink flow that is exploitable in practice. This significant auditing effort was carried out by hand, without using automated tools, to ensure accurate and unbiased classification of the reports. Note that the reports by JSA are strictly more refined than those output by the taint analysis, because JSA is able to provide subclassifications for detected vulnerabilities based not only on the involved source/sink pair but also the respective string values, as discussed in Section 5.1. The expert performing the review noted this as a major usability feature of JSA.

Results.

The results of the manual review are summarized in Figure 8. JSA reported a total of 2,940 vulnerabilities on 188 of the 675 websites, of which 301 were classified as false, compared with 7,087 reports by the taint analysis with close to 4,500 false findings. Importantly, not even a single true report by the taint analysis was missed by JSA, though in theory this could happen. The gain in accuracy is significant. Computed as the number of true reports divided by the number of overall reports, the accuracy of JSA is 0.9 compared with 0.37 for the baseline taint analysis.

As an illustration of JSA's limitations, we revisit the code in Listing 1 under Figure 7. This benign flow is misclassified by JSA as a vulnerability because JSA was unable to resolve `zipCodeRedirect` into its constant value. The reason is that `changeZipRedirect(...)` is invoked reflectively, which makes the resolution of `zipCodeRedirect` highly nontrivial. The conservative judgment made by JSA, given its inability to establish that the URL prefix is outside the user's control, is that the flow in Listing 1 is potentially vulnerable, which is a false alarm. Fortunately, as our statistics above indicate, cases like this are relatively rare, resulting in only

Listing 1: Code from `www.alltel.com`

```
function changeZipRedirect(zipCodeRedirect) {
 var currURL = document.location.href; ...; wcmContext = currURL.split('WCM_GLOBAL_CTXT'); var redirectStr=wcmContext[1]; ...;
 if ( redirectStr .match("pmapmc=") == null) {
  /* redirect to the zipcode page */ document.location.href=zipCodeRedirect + "&redirectURL" + redirectStr; } }
```

Listing 2: Code from `www.safeway.com`

```
var domainName="safeway";
var environment="www";
var sURL ="http://www.safeway.com/";
var prodURL="http://www.safeway.com/";
us=window.location.href;
usa=us.split(".");
for (i=0;i<usa.length;i++) {
 domainName=usa[1];
 environment=usa[0]; }
switch (domainName) {
 case "carrsqc":
  sURL =environment +"." + "carrsqc.com/" ;
  prodURL="http://www.carrsqc.com/";
  break;
 case "dominicks":
  sURL =environment +"." + "dominicks.com/" ;
  prodURL="http://www.dominicks.com/";
  break;
 ...
 default: sURL =environment +"." + "safeway.com/" ; prodURL="http://www.safeway.com/"; }
document.write('<a href=javascript:parent.location.href="'+sURL+'IFL/Grocery/T-O-U" title="Terms of Use">Terms of Use</a>');
```

Listing 3: Code from `http://www.corning.com`

```
var pageUrl = window.location;
var cId = document.getElementById("ctl00_ContentPlaceHolder1_hdnCId").value;
var url = "/CMS/OverviewPrint.aspx?id=" + cid + "&url=" + pageUrl;
openPopupWindow(url);
```

Listing 4: Code from IBM Team Concert Website

```
var url= window.location.href;
var i = url.indexOf("?");
if(i>0)
 url=url.substring(0, i);
 i = url.indexOf("/ntopic/");
if(i < 0)
 return;
url=url.substring(0, i+1);
url=url+"livehelp/?pluginID="+a;
window.location.href = url;
```

Figure 7: Several representative examples of JavaScript code from our benchmark suite

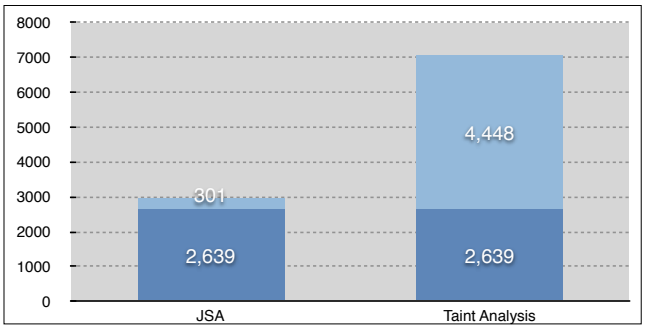

Figure 8: Breakdown of reported vulnerabilities by JSA and baseline taint analysis into true positives (bottom/dark) and false positives (top/pale)

10% false positives.

As for performance, both engines proved highly efficient. JSA required approximately 3 seconds on average to analyze a single webpage, where the taint analysis was (naturally) faster and completed in less than 2 seconds. There were only a few hundred exceptional cases that required significantly more than 3 seconds, but all cases terminated in under 30 seconds.

5.2.3 Comparison with Black-box Testing

The second experiment is complementary to the first experiment, which evaluates JSA against a purely static client-side analysis, in comparing between our hybrid approach and a purely dynamic testing algorithm for detection of client-side vulnerabilities. We compared between two AppScan configurations: The first has JSA enabled but the dynamic AppScan client-side tests disabled, and the second has JSA disabled and the AppScan client-side tests enabled, so we can contrast directly between JSA and and a commercial-grade client-side testing algorithm.

Methodology.

For the second experiment, we selected — at random — 60 out the original 675 websites. We then recreated the portion of each website that was retrieved by the crawler (up to 500 pages) locally to be able to perform testing without tampering with the production website. This time- and resource-consuming setup step is the reason why we restricted the second experiment to 60 out of the 675 websites, though we reemphasize that the sites were chosen at random. On each of the "local" websites, we ran the two AppScan configurations.

Results.

The results we obtained are summarized below:

Configuration	Vulnerable websites	False positives
JSA enabled	33	4
JSA disabled	8	0

Note that the numbers above are at the granularity of websites, and not specific vulnerabilities, to enable adequate comparison between the static and dynamic analyses. A website is counted under the "False positives" column if at least one of the reported findings for it is a false positive, where similarly to the first experiment, here too the findings were reviewed, classified and triaged manually by an expert.

The results clearly demonstrate the superiority of JSA in coverage. JSA detected true vulnerabilities in 29 websites compared to only 8 websites found vulnerable by the testing algorithm. Regarding false positives, which only JSA suffered from, expert review concluded that all 4 of the false reports refer to extremely complex, albeit feasible, path conditions (checking e.g. for browser vendor and version, installed plugins, etc), which means that JSA was technically correct though effectively the reported flows are unexploitable. We also note that the average performance of the dynamic client-site testing algorithm is in the order of 30-60 seconds per webpage, where JSA scans a webpage in under 3 seconds on average.

6. RELATED WORK

Numerous research approaches have been studied to detect security vulnerabilities in JavaScript code by using static and/or dynamic program analysis. Chugh, *et al.* [5] present a staged approach for handling the dynamic nature of JavaScript. First, static analysis is applied to as much of the information flow as possible based on the known code. Then, at the browser, residual checking is performed when new code is dynamically loaded. Our approach also performs hybrid analysis, with the difference that in our solution, we do not wait for the dynamic information to be automatically populated, but proactively make the concrete values referenced by the code available to a novel static string analysis.

Guha, *et al.* [14] use static analysis to extract a model of expected client behavior for JavaScript programs as seen from the server. This model is then used to build an intrusion prevention proxy for the server. To avoid mimicry attacks, random asynchronous requests are inserted. In a related study, Vogt, *et al.* [20] propose a system that stops XSS attacks already at the client by tracking flow of sensitive data inside the Web browser. If such information is about to be transferred to a third party, then the decision whether to permit the transfer is delegated to the user. JSA is fundamentally based on offline static analysis, applied to a partially evaluated JavaScript program, which obviates runtime overheads and has the potential to detect vulnerabilities that may not arise during a particular execution.

Maffeis, *et al.* [15] study three protection techniques against untrusted code to be executed in a trusted JavaScript environment: *filtering*, *rewriting* and *wrapping*. Filtering judges whether that code conforms to certain criteria; if not, then that code is rejected. Rewriting inserts run-time checks to inhibit undesirable actions by the untrusted code. Finally, wrapping protects sensitive resources of the trusted environment via run-time checks. Yu, *et al.* [23] use another form of rewriting to eliminate security attacks due to JavaScript, which identifies relevant operations within the JavaScript application, modifies suspicious behaviors, and prompts the user on how to proceed when appropriate. Our solution differs from the filtering phase of Maffeis, *et al.* because it augments the static analysis with dynamic data, which is essential for JavaScript and results in a novel string analysis. Also, JSA is an offline analysis for developers and auditors. Conversely, Maffeis, *et al.* and Yu, *et al.* target JavaScript run-time environments, and for this reason they augment the initial static analysis with rewriting and wrapping.

Gatekeeper, by Guarnieri and Livshits [12], detects security and reliability problems in JavaScript widgets. It uses a static pointer analysis to model the program's behavior, and issues queries against the pointer analysis to detect dangerous behaviors, even in the presence of malicious obfuscation. In a subsequent study, Guarnieri, *et al.* [13] present Actarus, a static-analysis tool for JavaScript that combines various static-analysis abstractions to model taint propagation. Their taint-tracking algorithm employs a custom context-sensitivity policy, and soundly models JavaScript constructs such as prototype-chain lookups and reflective property accesses. Compared to JSA, these algorithms are less accurate because (i) they use taint analysis, where JSA uses a form of string analysis, and (ii) the analysis is applied to the original HTML page (without partial evaluation).

More recently, Sridharan, *et al.* [17] identify correlated dynamic property accesses as a common code pattern that is traditionally analyzed imprecisely. They present a novel correlation tracking mechanism to cope with this pattern, thereby making the analysis more scalable. Scalability challenges remain, however, a problem. In a subsequent study, Feldthaus, *et al.* [11] address the problem of scaling the analysis to large JavaScript programs. They formulate a scalable, albeit unsound, field-based flow analysis for constructing call graphs. The resulting call graphs have general applicability, including for security analysis. These works address the important question of call-graph construction for JavaScript, which is orthogonal to our contributions. JSA could benefit from integration with more scalable and/or precise underlying analyses such as these.

Finally, Wei and Ryder [21] introduce a *blended* hybrid analysis approach for JavaScript programs. With their solution, dynamic traces containing information about method calls and object creation are fed as input to a static taint analysis, particularly to build a call graph that models code executed as the result of calls to `eval` and excludes code in uncovered branches. The number of arguments provided to methods calls is dynamically captured and used to partially relieve the loss of precision due to the fact that JavaScript functions are variadic. The static analysis phase is conservative, being a standard taint analysis, but at the same time it is also limited to the observed traces. Nevertheless, this work is a promising direction as a bug-finding tool for dynamic languages such as JavaScript.

7. CONCLUSION AND FUTURE WORK

We have presented JSA, a commercial security analysis for client-side JavaScript code that applies partial evaluation to JavaScript functions based on their dynamic HTML environment. This leads to a new form of string analysis, which we formalize and prove as sound and distributive. We also demonstrate, based on a comparative study over 170,000 webpages from top-popular websites, that JSA has clear advantages over traditional taint analysis as well as black-box client-side testing.

In the future, we intend to apply JSA to JavaScript code in mobile and server-side programs. We expect these new environments to yield different notions of partial evaluation, and consequently also new forms of string analysis. Another goal of our future research is to explore other sources of accuracy improvement thanks to partial evaluation, such as accounting for rare or complex path conditions (e.g., the common JavaScript test for browser version or plugin availability). We have observed that these are the main reason for the remaining false reports under JSA.

8. REFERENCES

[1] The dojo toolkit. `http://dojotoolkit.org/`.

[2] The jquery framework. `http://jquery.com/`.

[3] Watson libraries for analysis (wala). `https://github.com/wala/WALA`.

[4] The web 100 website. `http://www.web100.com/category/web-100/`.

[5] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged Information Flow for JavaScript. In *PLDI*, pages 50–62, 2009.

[6] R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. Dataflow analysis for concurrent programs using datarace detection. In *PLDI*, pages 316–326, 2008.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL '77*. ACM, 1977.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of POPL '79*. ACM, 1979.

[9] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13, 1992.

[10] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. 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*, 2010.

[11] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *ICSE*, pages 752–761, 2013.

[12] S. Guarnieri and B. Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security*, 2009.

[13] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *ISSTA*, pages 177–187, 2011.

[14] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *WWW*, pages 561–570, 2009.

[15] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with Filters, Rewriting and Wrappers. In *ESORICS*, 2009.

[16] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.

[17] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP*, pages 435–458, 2012.

[18] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE*, pages 210–225, 2013.

[19] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. In *PLDI*, pages 87–97, 2009.

[20] P. Vogt, F. Nentwich, N. Jovanovich, E. Kirda, C. Kruegel, and G. Vigna. Cross-site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, 2007.

[21] S. Wei and B. G. Ryder. Practical blended taint analysis for javascript. In *ISSTA*, pages 336–346, 2013.

[22] O. Weisman, L. Guy, O. Segal, and O. Tripp. Close encounters of the third kind. Technical report, Rational Brand, IBM Software Group, 2010.

[23] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *POPL*, 2007.