Precise and Scalable Static Analysis of jQuery using a Regular Expression Domain

Changhee Park
KAIST, Republic of Korea
changhee.park@kaist.ac.kr

Hyeonseung Im
Kangwon National University, Republic of Korea
hsim@kangwon.ac.kr

Sukyoung Ryu
KAIST, Republic of Korea
sryu.cs@kaist.ac.kr

Abstract

jQuery is the most popular JavaScript library but the state-of-the-art static analyzers for JavaScript applications fail to analyze simple programs that use jQuery. In this paper, we present a novel abstract string domain whose elements are simple regular expressions that can represent prefix, infix, and postfix substrings of a string and even their sets. We formalize the new domain in the abstract interpretation framework with abstract models of strings and objects commonly used in the existing JavaScript analyzers. For practical use of the domain, we present polynomial-time inclusion decision rules between the regular expressions and prove that the rules exactly capture the actual inclusion relation. We have implemented the domain as an extension of the open-source JavaScript analyzer, SAFE, and we show that the extension significantly improves the scalability and precision of the baseline analyzer in analyzing programs that use jQuery.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

Keywords JavaScript, static analysis, regular expressions

1. Introduction

Static analysis of JavaScript web applications often requires static analysis of jQuery [11] because it is the most popular JavaScript library. However, the dynamic natures of JavaScript that are heavily used in jQuery make static analysis impractical by causing scalability problems due to imprecise analysis results. For example, an imprecise approximate value of a property name `x` in a dynamic property access `obj[x]` may yield accessing all properties of `obj` at the worst case leading to the analysis state explosion.

To alleviate such problems in dynamic property accesses, static analysis should provide an elaborate string analysis model for property names, but existing analyzers use simple string domains that are not enough to represent first-class property names widely used in jQuery. Thus, the state-of-the-art JavaScript static analyzers such as JSAI [13], SAFE [19, 20], and TAJS [3] often fail to analyze simple applications that use jQuery. In such string domains, for instance, concatenation of a statically indeterminate value like a random value and a constant string is approximated as any string value, which may lead to the state explosion when a property access uses it as a property name.

In this paper, we present a novel string domain that consists of simple regular expressions. The domain is expressive enough to represent prefix, infix, and postfix substrings of a string and even their sets. We formalize the domain in abstract interpretation [6] for the core semantics of objects, and describe how it improves analysis precision in various abstract operations such as string concatenation and dynamic property read and write, compared to a constant string domain commonly used in existing JavaScript static analyzers.

For a sound abstraction, we define an order relation between regular expressions in the domain using the inclusion relation [8, 17] between them, which states that one regular expression R_1 includes another R_2 if a set of all strings that R_2 matches is a subset of the set of all strings that R_1 matches. While deciding the inclusion relation for general regular expressions has been known as an intractable problem [24], we present polynomial-time decision rules for the inclusion relation between the regular expressions in the domain and prove that the rules exactly capture the actual inclusion relation. Finally, we demonstrate that the new domain significantly improves the analysis scalability and precision of an existing JavaScript static analyzer.

The contributions of this paper are as follows:

- We formally describe how to extend an analysis model of string and object domains, commonly used in the existing static analyzers for JavaScript, with the new domain and prove its soundness theorem.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

DLS'16, November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4445-6/16/11...
http://dx.doi.org/10.1145/2989225.2989228

```
1  <script type="text/javascript"
2          src="jquery-1.7.0.js">
3  <ul id="ulUsers"></ul>
4  <script type="text/javascript">
5    $(function() {
6      $.get("http://tests.jquery-tutorial.net/
7                              json.php?callback=?",
8            function(data, textStatus) {
9              $.each(data, function(index, user){
10               $("#ulUsers").append(
11                 $("<li></li>").text(
12                     user.name + " is " +
13                     user.age + " years old"));
14             });
15           },
16           "json" );
17     });
18 </script>
```

Figure 1: An example from a jQuery tutorial

- We present polynomial-time inclusion decision rules for the regular expressions and prove their soundness and completeness with respect to the actual inclusion relation. To the best of our knowledge, no work has presented tractable inclusion decision rules for them.
- We have implemented the new domain as an extension of SAFE. We make the implementation of the extended SAFE open to the public [12].
- We evaluate the extension by analyzing 61 programs that use jQuery and show that the technique significantly improves the scalability and precision of SAFE.

The rest of the paper is organized as follows. Section 2 gives an example showing challenges in existing static analyzers, and informally describes the new domain. In Section 3, we formalize the concrete semantics and the existing analysis model of JavaScript strings and objects. Sections 4 and 5 formally present the regular expression domain and its inclusion rules, and Section 6 evaluates the domain with respect to analysis scalability and precision. Finally, we discuss related work in Section 7 and conclude in Section 8.

2. Motivation

This section shows an example code that significantly degrades analysis precision and scalability of a state-of-the-art JavaScript static analyzer and describes our solution.

2.1 Example

Figure 1 shows a simple program from a jQuery tutorial[1], which contains 18 lines of code that updates the contents in the `<ul>` tag (line 3) with data received from a server using an AJAX request. The program first loads a jQuery library (lines 1~2) and registers an anonymous function as a callback (line 5) to be called once a browser has completed loading the document; when this callback is called, using

[1] `http://www.jquery-tutorial.net/ajax/requesting-a-file-from-a-different-domain-using-jsonp`

```
1  expando: "jQuery" +
2           (jQuery.fn.jquery + Math.random()) ...
3  jsc = jQuery.now(), ...
4  jsonpCallback: function() {
5    return jQuery.expando + "_" + (jsc++);
6  } ...
7  jQuery.ajaxPrefilter("json jsonp",
8   function (...) { ...
9     jsonpCallback =  ... s.jsonpCallback() ...
10    previous = window[jsonpCallback], ...
11    window[jsonpCallback] = function() { ... }; ...
12    jqXHR.always(function() { ...
13       window[jsonpCallback] = previous; ... }) ...
14  });
```

Figure 2: An excerpt from jQuery 1.7.0

`$.get` (line 6) it receives JSON data from a server url at lines 6 and 7 and passes the data to the handler at lines 8~15; then the handler creates `<li>` tags with the user information inside the `<ul>` tag (lines 11~13). Hence, if a server provides the following JSON data,

```
[{"name":"John Doe","age":42},
 {"name":"Jane Doe","age":39}]
```

then, the program creates the following tags with the data as the result of the program:

```
<li> John Doe is 42 years old </li>
<li> Jane Doe is 39 years old </li>
```

2.2 Challenges in Static Analysis

Statically analyzing the `$` function in jQuery requires high analysis precision because it behaves differently for different argument values. For example, the `$` calls at lines 5, 10, and 11 in Figure 1 perform different tasks; the first registers its argument function as a load event handler, the second finds a DOM element with the `ulUsers` id, and the third creates a new DOM element with the `<li>` tag. Since SAFE supports flow-sensitive and context-sensitive analyses, it can analyze different behaviors of the `$` function with high precision.

However, SAFE fails to analyze the simple program in Figure 1 within the timeout of 10 hours. Figure 2 shows an excerpt from jQuery 1.7.0 where the anonymous function at line 8 is called to install a callback function with an appointed name before the actual AJAX request. In the case of the example in Figure 1, since the program does not appoint any name for a callback function by `"callback=?"` (line 7 of Figure 1), jQuery gives a random name by calling `jsonpCallback` (line 4), which returns a string that results from concatenating the values of `jQuery.expando`, `"_"`, and `jsc`; lines 1,2 and 3 show the definitions of `jQuery.expando` and `jsc` that have string values with a random number postfixed and a number representation of the current time, respectively. Then, it makes a backup for the old value with the name `jsonpCallback` (line 10), installs a new callback function (line 11), and restores the old value after the request is done (line 13).

Note that existing static analyzers for JavaScript applications approximate concatenation of a string and a statically indeterminate number string like the results of `Math.random()` and `jQuery.now()` as the string value that may become any string; from now on, we call such a value $\top_s$. Then, the analysis result of `jsonpCallback` at line 9 is $\top_s$, and that of `previous` at line 10 is a combined value of all global variable values including global functions, because the `window` object represents a global scope object in JavaScript and thus `window[jsonpCallback]` accesses all global variables. What is worse, this imprecise value is assigned to each global variable at line 13 polluting all variables at the global scope. We found that a subsequent global function call is resolved into 69 possible calls incurring a huge computation overhead during analysis to reach a fixpoint. The problem lies in that the JavaScript static analyzers use simple variants of the constant string lattice domain. Using such simple domains, analyzers often approximate the value of `jsonpCallback` as $\top_s$ losing the information from the concatenation that the value should include `"_"`.

A fundamental solution may be to use a more expressive domain but we found that all 12 string domains suggested by Madsen and Andreasen [16] are not sufficient to solve the problem. A character inclusion domain [16] can express the inclusion of `"_"` in `jsonpCallback`, but it cannot express `jQuery.expando` at line 1, which jQuery frequently uses as a key to property accesses. In this case, the prefix domain would be helpful since we can use the prefix `"jQuery"`. However, unlike the character inclusion domain, it cannot express the inclusion of `"_"` in `jsonpCallback`. All 12 string domains in the previous work [16] fail to express both information for `jsonpCallback` and `expando`.

2.3 Regular Expression Domain

As a solution, we present a novel abstract string domain that has regular expressions as its elements. The domain is expressive enough to represent prefix, infix, and postfix substrings and even their sets. It contains a special element $\star$, which is regarded as $\top_s$ but may be concatenated with other constant strings improving the expressiveness. For example, the abstract values for `jsonpCallback` and `expando` in the previous example have `"⋆_⋆"` and `"jQuery⋆"` in our domain, respectively, where $\star$ matches any strings. Then, when an analyzer uses the values as keys to property accesses, it accesses only such properties that match the given regular expressions. In this way, the analyzer, remaining sound, can increase not only the analysis precision but also its performance getting benefits from the improved precision. As for the example in Figure 1, when we apply the new domain to SAFE, it completes the analysis within 9 minutes with precise analysis results that approximate each global function call as one correct function call. In the next sections, we formalize the domain in abstract interpretation and show that it significantly improves the scalability and precision of an existing static analyzer in analyzing programs that use jQuery.

$$
\begin{array}{rl}
obj, o & \in \mathbf{Object} = \mathbf{String} \to \mathbf{Value} \\
\mathsf{PropRead} & \in \mathbf{Object} \times \mathbf{String} \to \mathbf{Value} \\
\mathsf{PropWrite} & \in \mathbf{Object} \times \mathbf{String} \times \mathbf{Value} \to \mathbf{Object}
\end{array}
$$

$$
\mathsf{PropRead}(obj, x) = \begin{cases} obj(x) & \text{if } x \in dom(obj) \\ \mathsf{undefined} & \text{otherwise} \end{cases}
$$

$$
\mathsf{PropWrite}(obj, x, v) = \begin{cases} (obj \setminus [x \mapsto obj(x)]) * [x \mapsto v] & \text{if } x \in dom(obj) \\ obj * [x \mapsto v] & \text{otherwise} \end{cases}
$$

Figure 3: JavaScript objects and object property read/write

$$
\begin{array}{l}
\mathsf{PropReadSet} \in \wp(\mathbf{Object}) \times \wp(\mathbf{String}) \to \wp(\mathbf{Value}) \\
\mathsf{PropWriteSet} \\
\quad \in \wp(\mathbf{Object}) \times \wp(\mathbf{String}) \times \wp(\mathbf{Value}) \to \wp(\mathbf{Object}) \\
\mathsf{StrConcatSet} \in \wp(\mathbf{String}) \times \wp(\mathbf{String}) \to \wp(\mathbf{String})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{PropReadSet}(oset, sset) \\
\quad = \{\mathsf{PropRead}(o, x) \mid o \in oset \wedge x \in sset\} \\
\mathsf{PropWriteSet}(oset, sset, vset) \\
\quad = \{\mathsf{PropWrite}(o, x, v) \mid o \in oset \wedge x \in sset \wedge v \in vset\} \\
\mathsf{StrConcatSet}(sset_1, sset_2) \\
\quad = \{s_1 \oplus s_2 \mid s_1 \in sset_1 \wedge s_2 \in sset_2\}
\end{array}
$$

Figure 4: Property read/write operations on objects and string concatenation in collecting semantics

3. Formal Description

We formally present a simplified semantics of property read and write operations of objects in JavaScript and describe an existing static analysis model for them, which are commonly used in JSAI, SAFE, and TAJS. For presentation brevity, we omit internal properties in objects and prototype-related parts in the formalization. Note that our implementation addresses them as well using the existing support from the baseline SAFE analyzer.

3.1 Concrete Semantics of JavaScript Objects

Figure 3 presents definitions of JavaScript objects and their property read and write operations using the following:

- $dom(obj)$: returns a set of all domain elements of obj
- $obj * [x \mapsto v]$: adds a map element $[x \mapsto v]$ to obj
- $obj \setminus [x \mapsto v]$: deletes a map element $[x \mapsto v]$ from obj.

An object maps strings to values, $\mathbf{Value}$, and the property read and write operations are functions from a pair of an object and a string to a value (PropRead) and from a tuple of an object, a string, and a value to an updated object (PropWrite), respectively. Note that, in JavaScript, an absent property read gives the undefined value and an absent property write adds a new property to the target object. Figure 4 shows lifted property read and write operations for objects and string concatenation in collecting semantics; we use $\wp(S)$ for the power set of the set S and $\oplus$ for string con-

$$
\begin{array}{ll}
\text{(abstraction)} & \alpha_s \in \wp(\mathbf{String}) \mapsto \hat{\mathbf{String}} \\
\text{(concretization)} & \gamma_s \in \hat{\mathbf{String}} \mapsto \wp(\mathbf{String})
\end{array}
$$

$$
\alpha_s(\mathit{sset}) = \begin{cases} \bot_s & \text{if } \mathit{sset} = \{\} \\ x & \text{if } \mathit{sset} = \{x\} \\ \top_s & \text{otherwise} \end{cases}
$$

$$
\gamma_s(\hat{s}) = \begin{cases} \{\} & \text{if } \hat{s} = \bot_s \\ \{x\} & \text{if } \hat{s} = x \\ \mathbf{String} & \text{if } \hat{s} = \top_s \end{cases}
$$

$$
\hat{\mathsf{StrConcat}}(\hat{s}_1, \hat{s}_2) = \begin{cases} \bot_s & \text{if } \hat{s}_1 = \bot_s \vee \hat{s}_2 = \bot_s \\ x_1 \oplus x_2 & \text{if } \gamma_s(\hat{s}_1) = \{x_1\} \wedge \gamma_s(\hat{s}_2) = \{x_2\} \\ \top_s & \text{otherwise} \end{cases}
$$

Figure 5: Galois connection of strings and abstract string concatenation

catenation. The collecting semantics approximates all possible result values of operations on a given program point in all reachable execution traces. For example, on a certain program point with a PropRead operation, if possible input values for PropRead in all execution are two objects, $[a \mapsto 1]$ and $[b \mapsto 2]$, and two property names, "a" and "b", then PropReadSet produces $\{1, 2, \mathsf{undefined}\}$ that contains result values from all combinations of the input values. Throughout this paper, we present only the string concatenation $\oplus$ as a representative string operation.

3.2 Static Analysis Model

We present an analysis model of strings and objects commonly used in existing analyzers, and we formalize them in the abstract interpretation framework [6].

Abstract strings. An abstract string domain denoted by $\hat{\mathbf{String}}$ is the constant string lattice domain whose element is either $\top_s$, a constant string, or $\bot_s$; intuitively, $\top_s$ means any string, a constant means a single concrete string, and $\bot_s$ no string. The elements have the following order relation $\sqsubseteq_s$:

$$\hat{s}_1 \sqsubseteq_s \hat{s}_2 \Leftrightarrow (\hat{s}_1 = \bot_s \ \vee \ \hat{s}_2 = \top_s \ \vee \ \hat{s}_1 = \hat{s}_2).$$

The domain is the one that JSAI, SAFE, and TAJS commonly support while they provide variants of it. All of them have additional elements that represent any number strings and any non-number strings to distinguish property names for arrays, JSAI has another element to represent built-in property names of JavaScript objects. We omit the details of the variants because they still have the same problem described in Section 2.2 as the base constant domain does. Figure 5 shows the definitions of two translation functions α_s and γ_s. It is trivial to show that these two functions satisfy the following Galois connection condition:

THEOREM 1. *(Galois connection of strings)*
$\forall s \in \wp(\mathbf{String}), \hat{s} \in \hat{\mathbf{String}} : \alpha_s(s) \sqsubseteq_s \hat{s} \Longleftrightarrow s \subseteq \gamma_s(\hat{s})$.

$$
\begin{array}{lcl}
\bot_o,\ \hat{o} = \langle \mathit{propmap}, \hat{\mathit{sum}}, \mathit{defset} \rangle \in \hat{\mathbf{Object}} & = & \\
\quad \{\bot_o\} \cup ((\mathbf{String} \mapsto \hat{\mathbf{Value}}) \times \hat{\mathbf{Value}} \times \wp(\mathbf{String})) & & \\
\alpha_v & \in & \wp(\mathbf{Value}) \to \hat{\mathbf{Value}} \\
\gamma_v & \in & \hat{\mathbf{Value}} \to \wp(\mathbf{Value}) \\
\alpha_o & \in & \wp(\mathbf{Object}) \to \hat{\mathbf{Object}} \\
\gamma_o & \in & \hat{\mathbf{Object}} \to \wp(\mathbf{Object})
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{Propmap}(\langle \mathit{propmap}, \hat{\mathit{sum}}, \mathit{defset} \rangle) & = & \mathit{propmap} \\
\mathsf{Summary}(\langle \mathit{propmap}, \hat{\mathit{sum}}, \mathit{defset} \rangle) & = & \hat{\mathit{sum}} \\
\mathsf{Defset}(\langle \mathit{propmap}, \hat{\mathit{sum}}, \mathit{defset} \rangle) & = & \mathit{defset}
\end{array}
$$

$$
\begin{array}{l}
\hat{o}_1 \sqsubseteq_o \hat{o}_2 \Leftrightarrow (\hat{o}_1 = \bot_o \ \vee \ (\hat{o}_1 \neq \bot_o \ \wedge \ \hat{o}_2 \neq \bot_o \ \wedge \\
\quad \bigsqcup_v \{\mathsf{Propmap}(\hat{o}_1)(s) \mid \\
\qquad s \in \mathit{dom}(\mathsf{Propmap}(\hat{o}_1)) \setminus \mathit{dom}(\mathsf{Propmap}(\hat{o}_2))\} \\
\quad \sqsubseteq_v \mathsf{Summary}(\hat{o}_2) \\
\quad \wedge \ \forall s \in (\mathit{dom}(\mathsf{Propmap}(\hat{o}_1)) \cap \mathit{dom}(\mathsf{Propmap}(\hat{o}_2))) : \\
\qquad \mathsf{Propmap}(\hat{o}_1)(s) \sqsubseteq_v \mathsf{Propmap}(\hat{o}_2)(s) \\
\quad \wedge \ \mathsf{Summary}(\hat{o}_1) \sqsubseteq_v \mathsf{Summary}(\hat{o}_2) \\
\quad \wedge \ \mathsf{Defset}(\hat{o}_2) \subseteq \mathsf{Defset}(\hat{o}_1)))
\end{array}
$$

$$
\hat{o}_1 \sqcup_o \hat{o}_2 = \begin{cases} \text{if } \hat{o}_1 = \bot_o, \quad \hat{o}_2 \\ \text{if } \hat{o}_2 = \bot_o, \quad \hat{o}_1 \\ \text{otherwise,} \\ \quad \langle \ \{[s \mapsto (\mathsf{Propmap}(\hat{o}_1)(s) \sqcup_v \mathsf{Propmap}(\hat{o}_2)(s))] \mid \\ \quad s \in \mathit{dom}(\mathsf{Propmap}(\hat{o}_1)) \ \vee \ s \in \mathit{dom}(\mathsf{Propmap}(\hat{o}_2))\}, \\ \quad \mathsf{Summary}(\hat{o}_1) \sqcup_v \mathsf{Summary}(\hat{o}_2), \\ \quad \mathsf{Defset}(\hat{o}_1) \cap \mathsf{Defset}(\hat{o}_2) \ \rangle \end{cases}
$$

$$
\alpha_o(\mathit{oset}) = \begin{cases} \text{if } \mathit{oset} = \{\}, \quad \bot_o \\ \text{otherwise,} \\ \quad \langle \ \{[s \mapsto \alpha_v(\{o_2(s) \mid o_2 \in \mathit{oset} \wedge s \in \mathit{dom}(o_2)\})] \\ \qquad\qquad \mid o \in \mathit{oset} \wedge s \in \mathit{dom}(o)\}, \\ \quad \bot_v, \bigcap\{\mathit{dom}(o) \mid o \in \mathit{oset}\} \ \rangle \end{cases}
$$

$$
\gamma_o(\hat{o}) = \bigcup\{\mathit{oset} \mid \alpha_o(\mathit{oset}) \sqsubseteq \hat{o}\}
$$

Figure 6: Abstract objects and Galois connection of objects

$\hat{\mathsf{StrConcat}}$ in Figure 5 defines the abstract string concatenation on the abstract string domain. Note that while this operation gives a precise result only when both of two arguments indicate constant string values, it returns the imprecise $\top_s$ value when either of two arguments is $\top_s$. Then the imprecise value may flow into a key for property accesses in objects and incur a huge performance overhead due to the state explosion such as the one described in Section 2. We can prove that the function $\hat{\mathsf{StrConcat}}$ is monotone and a sound approximation of StrConcatSet. Formal theorems and their proofs are publicly available [12].

Abstract objects. Figure 6 shows the definition of abstract objects. An abstract object $\hat{o}$ is either a bottom object ($\bot_o$) or a tuple of a property map (*propmap*), a summary abstract value ($\hat{\mathit{sum}}$), and a definite set of strings (*defset*):

- *propmap* : a map from property names to abstract values; it may have mappings for absent properties.
- *sûm* : an approximate value for properties that may be in the current object; for example, $\hat{o}[\top_s]$ reads this value and $\hat{o}[\top_s] = \hat{v}$ updates this value instead of adding new map elements from all strings to $\hat{v}$ in *propmap*
- *defset* : a set of property names that the current object definitely should include.

The helper functions, Propmap, Summary, and Defset, are accessors to the components of an abstract object. Figure 6 also shows the definitions of an order relation ($\sqsubseteq_o$) and a join operation ($\sqcup_o$) between abstract objects. We abuse set operations for property maps by regarding them as sets of map elements and assume that a set of abstract values, $\mathbf{Val\hat{u}e}$, is defined as a complete lattice with the order relation $\sqsubseteq_v$ and the join operation $\sqcup_v$.

The α_o and γ_o functions define the Galois connection of objects; α_o abstracts a concrete object set (*oset*) to an abstract object; if *oset* is an empty set, then α_o abstracts it as $\bot_o$ and otherwise, α_o abstracts it as a tuple whose property map is a set of map elements from all property names in objects in *oset* to abstract values that approximate concrete property values with the same property names, whose summary value is the bottom value $\bot_v$, and whose definite set of strings is the intersection of the sets of property names in the objects in *oset*. We assume the Galois connection on values, α_v and γ_v, and prove that α_o and γ_o satisfy the Galois connection:

THEOREM 2. *(Galois connection of objects)*
$\forall o \in \wp(\mathbf{Object}), \hat{o} \in \mathbf{\hat{Object}} : \alpha_o(o) \sqsubseteq_o \hat{o} \iff o \subseteq \gamma_o(\hat{o})$.

Note that the Galois connection conditions that we showed in Theorem 1 and Theorem 2 are one of the requirements in abstract interpretation to guarantee the soundness of the analysis that uses the domain and they are used to prove the soundness of other operations in the domain.

Abstract property read and write operations. Figure 7 shows the definitions of abstract property read ($\hat{\mathsf{PropRead}}$) and write ($\hat{\mathsf{PropWrite}}$) operations. When reading a property named $\top_s$, $\hat{\mathsf{PropRead}}$ joins all property values and the $\hat{\mathsf{undefined}}$ value, and returns the joined value (the first rule) because the property name may be any string and thus it may extract any property value as well as the undefined value; if the property name is a constant name and the property map of the current object has a mapping for the name, depending on whether the object *definitely* has the property or not, the function returns the value from the property map (the second rule) or returns the value joined with the summary value and $\hat{\mathsf{undefined}}$ (the third rule); if the property name is a constant name but it is not in the property map (the fourth rule), the function returns a join of the summary value and $\hat{\mathsf{undefined}}$ since the property with the given name may have the summary value; in other cases, the function returns $\bot_v$.

$$\hat{\mathsf{PropRead}}(\hat{o}, \hat{s}) \in \mathbf{\hat{Object}} \times \mathbf{\hat{String}} \to \mathbf{\hat{Value}}$$

$$= \begin{cases} \text{if } \hat{s} = \top_s, \\ \quad \bigsqcup_v \{\mathsf{Propmap}(\hat{o})(s) \mid s \in \mathit{dom}(\mathsf{Propmap}(\hat{o}))\} \\ \quad \sqcup_v \mathsf{Summary}(\hat{o}) \sqcup_v \hat{\mathsf{undefined}} \\ \text{if } \gamma_s(\hat{s}) = \{s\} \wedge s \in \mathit{dom}(\mathsf{Propmap}(\hat{o})) \\ \qquad\qquad \wedge\ s \in \mathsf{Defset}(\hat{o}), \\ \quad \mathsf{Propmap}(\hat{o})(s) \\ \text{if } \gamma_s(\hat{s}) = \{s\} \wedge s \in \mathit{dom}(\mathsf{Propmap}(\hat{o})) \\ \qquad\qquad \wedge\ s \notin \mathsf{Defset}(\hat{o}), \\ \quad \mathsf{Propmap}(\hat{o})(s) \sqcup_v \mathsf{Summary}(\hat{o}) \sqcup_v \hat{\mathsf{undefined}} \\ \text{if } \gamma_s(\hat{s}) = \{s\} \ \wedge\ s \notin \mathit{dom}(\mathsf{Propmap}(\hat{o})), \\ \quad \mathsf{Summary}(\hat{o}) \sqcup_v \hat{\mathsf{undefined}} \\ \text{otherwise, } \bot_v \end{cases}$$

$$\hat{\mathsf{PropWrite}}(\hat{o}, \hat{s}, \hat{v}) \in \mathbf{\hat{Object}} \times \mathbf{\hat{String}} \times \mathbf{\hat{Value}} \to \mathbf{\hat{Object}}$$

$$= \begin{cases} \text{if } \hat{s} = \top_s, \\ \quad \langle\ \{[s \mapsto (\hat{v} \sqcup_v \hat{v}_2)] \mid s \in \mathit{dom}(\mathsf{Propmap}(\hat{o})) \\ \qquad\qquad \wedge\ \hat{v}_2 = \mathsf{Propmap}(\hat{o})(s)\}, \\ \qquad \mathsf{Summary}(\hat{o}) \sqcup_v \hat{v},\ \mathsf{Defset}(\hat{o})\ \rangle \\ \text{if } \gamma_s(\hat{s}) = \{s\} \wedge s \in \mathit{dom}(\mathsf{Propmap}(\hat{o})) \\ \qquad\qquad \wedge\ s \in \mathsf{Defset}(\hat{o}), \\ \quad \langle\ (\mathsf{Propmap}(\hat{o}) \setminus [s \mapsto \mathsf{Propmap}(\hat{o})(s)]) * [s \mapsto \hat{v}], \\ \qquad\qquad \mathsf{Summary}(\hat{o}),\ \mathsf{Defset}(\hat{o})\ \rangle \\ \text{if } \gamma_s(\hat{s}) = \{s\} \wedge s \in \mathit{dom}(\mathsf{Propmap}(\hat{o})) \\ \qquad\qquad \wedge\ s \notin \mathsf{Defset}(\hat{o}), \\ \quad \langle\ (\mathsf{Propmap}(\hat{o}) \setminus [s \mapsto \mathsf{Propmap}(\hat{o})(s)]) \\ \qquad * [s \mapsto \hat{v}],\ \mathsf{Summary}(\hat{o}),\ \mathsf{Defset}(\hat{o}) \cup \{s\}\ \rangle \\ \text{if } \gamma_s(\hat{s}) = \{s\} \wedge s \notin \mathit{dom}(\mathsf{Propmap}(\hat{o})), \\ \quad \langle\ \mathsf{Propmap}(\hat{o}) * [s \mapsto \hat{v}], \\ \qquad\qquad \mathsf{Summary}(\hat{o}),\ \mathsf{Defset}(\hat{o}) \cup \{s\}\ \rangle \\ \text{otherwise, } \bot_o \end{cases}$$

Figure 7: Property read/write operations on abstract objects

When writing a property with some value to an object, if the property name is $\top_s$ (the first rule), $\hat{\mathsf{PropWrite}}$ *weakly* updates all the property values and the summary value; if the property name is a constant and the property map of the object already has the property (the second and third rules), the functions *strongly* update the property map of the object with the new property information. Note that the third rule updates the definite set of property names with the new name stating that the updated property name now is definitely present in the object. If the property name is a constant but the property map of the object does not have the property (the fourth rule), it adds the property information to the property map with the new value and updates the definite set of property names to include the new name; in other cases, it returns the bottom object ($\bot_o$) meaning no object.

We prove that $\hat{\mathsf{PropRead}}$ and $\hat{\mathsf{PropWrite}}$ are monotone and sound with respect to PropReadSet and PropWriteSet, respectively. For presentation brevity, we present only the

soundness theorem here and make the complete formal theorems and their proofs publicly available [12].

THEOREM 3. *(Soundness of* $\hat{\mathsf{PropRead}}$ *and* $\hat{\mathsf{PropWrite}}$*)*
$(\forall o \in \wp(\mathbf{Object}), s \in \wp(\mathbf{String}) :$
$\quad \alpha_v(\mathsf{PropReadSet}(o,s)) \sqsubseteq_v \hat{\mathsf{PropRead}}(\alpha_o(o), \alpha_s(s)))$
$\wedge$
$(\forall o \in \wp(\mathbf{Object}), s \in \wp(\mathbf{String}), v \in \wp(\mathbf{Value}) :$
$\quad \alpha_o(\mathsf{PropWriteSet}(o,s,v))$
$\quad \sqsubseteq_o \hat{\mathsf{PropWrite}}(\alpha_o(o), \alpha_s(s), \alpha_v(v))).$

4. Regular Expression Domain

Now, we present a novel and sound string domain that uses regular expressions and show how to extend the string concatenation and property read and write operations defined in the previous section to the ones in the new domain.

Regular expression elements. We first define the syntax of an atomic regular expression, $A \in \mathbb{A}$, as follows:

$$A ::= \texttt{Empty} \mid \star \mid aA \mid \star aA$$
where $a \in \Sigma$ (a set of alphabets).

The domain $\mathbb{A}$ is a set of all atomic regular expressions that can be produced by the production rules above. Note that $\star$ is a special character, which is not a member of the alphabet set ($\star \notin \Sigma$) and can match any string. Then, our regular expression domain $\mathbb{R}$ is a domain of atomic regular expression sets with the following condition:

$$\forall R \in \mathbb{R} : \mathsf{getAlphabetSet}(R) \subsetneq \Sigma$$

where

$$\begin{array}{rcl}
\mathsf{getAlphabetSet}(R) & = & \bigcup_{A \in R} \mathsf{getAlphabets}(A) \\
\mathsf{getAlphabets}(A) & = & \{\} \quad \text{if } A = \texttt{Empty} \vee A = \star \\
\mathsf{getAlphabets}(aA) & = & \{a\} \cup \mathsf{getAlphabets}(A) \\
\mathsf{getAlphabets}(\star aA) & = & \{a\} \cup \mathsf{getAlphabets}(A)
\end{array}$$

A regular expression $R \in \mathbb{R}$ semantically represents disjunctions over all atomic regular expressions in R. The getAlphabetSet function simply collects all alphabets that appear in a given regular expression and the restriction states that the union of all alphabets that appear in our regular expressions does not make the total set of alphabets. We found that this restriction makes inclusion decision rules between our regular expressions simpler and faster as a polynomial-time problem in the worst case than the one without the restriction [7] that has the exponential-time complexity. We explain our inclusion decision rules in the next section.

Semantics. The result of the semantic function $[\![R]\!]$ with a regular expression R is defined as a set of all strings that R matches and the following shows the formal definitions:

$$\begin{array}{rcl}
[\![\{\}]\!] & = & \{\} \\
[\![R]\!] & = & \bigcup_{A \in R} [\![\{A\}]\!] \qquad \text{if } |R| \geq 2 \\
[\![\{\texttt{Empty}\}]\!] & = & \{\epsilon\} \qquad \text{where } \epsilon \text{ is an empty string} \\
[\![\{\star\}]\!] & = & \mathbf{String} \\
[\![\{aA\}]\!] & = & \{as \mid s \in [\![\{A\}]\!]\} \\
[\![\{\star aA\}]\!] & = & \{s_1 s_2 \mid s_1 \in \mathbf{String} \wedge s_2 \in [\![\{aA\}]\!]\}
\end{array}$$

$$\hat{\mathsf{NewStrConcat}}(R_1, R_2) = \{\ A_1A_2 \mid A_1 \in R_1 \wedge A_2 \in R_2\ \}$$

$$\hat{\mathsf{NewPropRead}}(\hat{o}, R) = \begin{cases}
\text{if } \hat{o} = \bot_o \vee R = \{\},\ \bot_v \\
\text{if } [\![R]\!] \text{ is a finite set with } |[\![R]\!]| \geq 1, \\
\quad \bigsqcup_v \{\hat{\mathsf{PropRead}}(\hat{o}, \alpha_s(\{s\})) \mid s \in [\![R]\!]\} \\
\text{otherwise,} \\
\quad \bigsqcup_v \{\mathsf{Propmap}(\hat{o})(s) \mid s \in dom(\mathsf{Propmap}(\hat{o})) \\
\qquad\qquad \wedge\ s \in [\![R]\!]\} \\
\quad \sqcup_v \mathsf{Summary}(\hat{o}) \sqcup_v \hat{\mathsf{undefined}}
\end{cases}$$

$$\hat{\mathsf{NewPropWrite}}(\hat{o}, R, \hat{v}) = \begin{cases}
\text{if } \hat{o} = \bot_o \vee R = \{\} \vee \hat{v} = \bot_v,\ \bot_o \\
\text{if } [\![R]\!] \text{ is a finite set with } |[\![R]\!]| \geq 1, \\
\quad \bigsqcup_o \{\hat{\mathsf{PropWrite}}(\hat{o}, \alpha_s(\{s\}), \hat{v}) \mid s \in [\![R]\!]\} \\
\text{otherwise,} \\
\quad \langle\ \{[s \mapsto \mathsf{Propmap}(\hat{o})(s)] \mid s \in dom(\mathsf{Propmap}(\hat{o})) \\
\qquad\qquad \wedge\ s \notin [\![R]\!]\} \\
\qquad \cup\ \{[s \mapsto (\hat{v} \sqcup_v \hat{v}_2)] \mid s \in dom(\mathsf{Propmap}(\hat{o})) \\
\qquad\qquad \wedge\ s \in [\![R]\!] \wedge \hat{v}_2 = \mathsf{Propmap}(\hat{o})(s)\}, \\
\qquad\qquad \mathsf{Summary}(\hat{o}) \sqcup_v \hat{v},\ \mathsf{DefSet}(\hat{o})\ \rangle
\end{cases}$$

Figure 8: String concatenation, property read and write on the regular expression domain

If R is an empty set, because it does not match any string, the function returns an empty string set; other definitions are by the usual definitions of disjunctions and conjunctions of regular expressions except that Empty matches only the empty string ϵ and $\star$ matches all strings.

Galois connection. We make our regular expression domain $\mathbb{R}$ as a lattice with the order relation $\sqsubseteq_r$, which is defined using the *inclusion relation* [8, 17] between regular expressions as follows:

$$R_1 \sqsubseteq_r R_2 \iff [\![R_1]\!] \subseteq [\![R_2]\!]$$

which means that the set of strings that R_1 matches is a subset of the set of strings that R_2 matches. Note that, while deciding the inclusion relation between general regular expressions has been known as an intractable problem with the PSPACE-complete complexity [24], we present novel decision rules that decide the inclusion relation between our regular expressions with the polynomial-time complexity in the next section. Now, we define the γ_r function to translate a regular expression to a set of concrete strings using the semantic function and the α_r function to translate a set of concrete strings to a regular expression using γ_r as follows:

$$\begin{array}{rcl}
\gamma_r(R) & = & [\![R]\!] \\
\alpha_r(sset) & = & \bigsqcap_r \{R \mid sset \subseteq \gamma_r(R)\}
\end{array}$$

We prove the α_r and γ_r satisfy the Galois connection:

THEOREM 4. *(Galois connection of regular expressions)*
$\forall s \in \wp(\mathbf{String}), R \in \mathbb{R} : \alpha_r(s) \sqsubseteq_r R \iff s \subseteq \gamma_r(R).$

Abstract operations. Note that all elements in the constant string domain $\hat{\mathbf{String}}$ that we defined in the previous section

can be represented using regular expressions in our new domain. Hence, all abstract operations defined in $\widehat{\mathbf{String}}$ are re-usable in the new regular expression domain and it is possible to refine the operations incrementally as needed. This point can save huge amounts of modification in the design and implementations of the existing analyzers that use constant string domains to use the new domain.

Figure 8 shows re-defined operations in the new domain where a regular expression matches a finite number of strings unless it contains $\star$. $\widehat{\mathsf{NewStrConcat}}$ simply concatenates all atomic regular expressions in input regular expressions; in this way, it can maintain substring information even when it concatenates a random string and a constant string unlike $\widehat{\mathsf{StrConcat}}$ that loses the information. $\widehat{\mathsf{NewPropRead}}$ returns no value ($\bot_v$) when the input values include no object ($\bot_o$) or no regular expression ($\{\}$) in the first rule and re-uses $\widehat{\mathsf{PropRead}}$ when the set of strings that the given regular expression matches is non-empty and finite. The function in the last rule retrieves all property values whose names match the given regular expression and soundly joins all the values with the summary value of the input object and $\widehat{\mathsf{undefined}}$. Similarly, $\widehat{\mathsf{NewPropWrite}}$ returns no object ($\bot_o$) with abnormal input values and re-uses $\widehat{\mathsf{PropWrite}}$ when the set of strings that the given regular expression matches is non-empty and finite. In the other case, it *weakly* updates the values of the properties whose names match the regular expression ($s \in [\![R]\!]$). Note that for the properties whose names do not match the regular expression ($s \notin [\![R]\!]$), it keeps the original values ($\mathsf{Propmap}(\hat{o})(s)$).

The operations in Figure 8 are monotone and sound approximation of their corresponding concrete operations:

THEOREM 5. *(Soundness of* $\widehat{\mathsf{NewPropRead}}$ *and* $\widehat{\mathsf{NewPropWrite}}$*)*

$(\forall o \in \wp(\mathbf{Object}), s \in \wp(\mathbf{String}) :$
$\alpha_v(\mathsf{PropReadSet}(o, s)) \sqsubseteq_v \widehat{\mathsf{NewPropRead}}(\alpha_o(o), \alpha_r(s)))$
$\wedge$
$(\forall o \in \wp(\mathbf{Object}), s \in \wp(\mathbf{String}), v \in \wp(\mathbf{Value}) :$
$\alpha_o(\mathsf{PropWriteSet}(o, s, v))$
$\sqsubseteq_o \widehat{\mathsf{NewPropWrite}}(\alpha_o(o), \alpha_r(s), \alpha_v(v))).$

The proof is publicly available [12].

5. Inclusion Decision Rules for Regular Expressions

Recall that we define the order relation $\sqsubseteq_r$ between our regular expressions using the inclusion relation as follows:

$$R_1 \sqsubseteq_r R_2 \iff [\![R_1]\!] \subseteq [\![R_2]\!].$$

Implementing an inclusion decision algorithm is necessary to practically use regular expression domains in sound static analysis, but as we mentioned in the previous section, the decision problem for general regular expressions is intractable. While the literature [8, 17] discovered some tractable cases with restricted subsets of regular expressions, we present novel decision rules for another subset of regular expressions, which has the polynomial-time complexity.

Auxiliary Functions

$$\begin{array}{lcl} \mathsf{pruneNonStar}(R) & = & \{\star aA \mid \star aA \in R\} \\ \mathsf{pruneHead}(a,\ R) & = & \mathsf{pruneNonStar}(R) \cup \\ & & \{A \mid aA \in R \ \vee\ \star aA \in R\} \end{array}$$

Inclusion Decision Rules

$R_1 \leq_r R_2$ means R_2 includes R_1.

$$\textsc{InclStar}\ \frac{\star \in R_2}{R_1 \leq_r R_2} \qquad \textsc{InclEmptySet}\ \frac{\star \notin R}{\{\} \leq_r R}$$

$$\textsc{InclLSet}\ \frac{\star \notin R_2 \quad |R_1| \geq 2 \quad \forall A \in R_1 : \{A\} \leq_r R_2}{R_1 \leq_r R_2}$$

$$\textsc{InclEmpty}\ \frac{\star \notin R \quad \mathtt{Empty} \in R}{\{\mathtt{Empty}\} \leq_r R}$$

$$\textsc{InclAReg}\ \frac{\star \notin R \quad \{A\} \leq_r \mathsf{pruneHead}(a,\ R)}{\{aA\} \leq_r R}$$

$$\textsc{InclStarReg}\ \frac{\star \notin R \quad \{aA\} \leq_r \mathsf{pruneNonStar}(R)}{\{\star aA\} \leq_r R}$$

Figure 9: Inclusion decision rules for regular expressions in $\mathbb{R}$

Figure 9 shows the definition of the decision relation $\leq_r$. Let us consider core rules first using the $\{\star aa\} \leq_r \{\star a, b\}$ example. The INCLSTARREG rule prunes the atomic regular expression b in $\{\star a, b\}$ using pruneNonStar because b can never include $\star aa$, and then it tries to decide $\{aa\} \leq_r \{\star a\}$ since if $\star a$ includes aa, it also includes $\star aa$. Next, the INCLAREG rule matches the first character a in aa with $\star a$ and removes the matched parts in them, and then the decision depends on that of the remaining parts, $\{a\} \leq_r \{\star a, \mathtt{Empty}\}$. Note that pruning the matched part a in $\{\star a\}$ by pruneHead produces $\{\star a, \mathtt{Empty}\}$ because $\{\star a\}$ is semantically equivalent to $\{\top \star a, a\}$ if $\top$ means a regular expression that matches any single character, and pruning a in $\{\top \star a, a\}$ produces $\{\star a, \mathtt{Empty}\}$. Likewise, the INCLAREG rule re-applies to $\{a\} \leq_r \{\star a, \mathtt{Empty}\}$ again and prunes the head a in both $\{a\}$ and $\{\star a, \mathtt{Empty}\}$ leaving the decision $\{\mathtt{Empty}\} \leq_r \{\star a, \mathtt{Empty}\}$. Finally, the INCLEMPTY rule accepts the decision since $\mathtt{Empty} \in \{\star a, \mathtt{Empty}\}$.

The rest are trivial. The INCLSTAR rule states that a regular expression that contains $\star$ includes any regular expression, and the other 5 rules handle the cases where R_2 in $R_1 \leq_r R_2$ does not have $\star$. The INCLEMPTYSET and INCLLSET rules are when the size of R_1 in $R_1 \leq_r R_2$ is 0 and bigger than 1, respectively; the former states that any regular expression includes no regular expression ($\{\}$) and the latter states that if R_2 includes all atomic regular expressions in R_1, it includes R_1 as well. Note that $\{\star\} \leq_r R_2$ does not hold unless R_2 contains $\star$. This is because of the restriction, $\forall R \in \mathbb{R} : \mathsf{getAlphabetSet}(R) \subsetneq \Sigma$, that we defined in

the previous section; thanks to the restriction, it is impossible without $\star$ to make syntactically different but semantically equivalent regular expressions with $\{\star\}$. We found that the restriction makes the decision rules and related proofs simpler with the smaller time complexity than without the restriction. The decision rules exactly capture the inclusion relation between regular expressions in the domain:

THEOREM 6. *(Soundness and completeness of $R_1 \leq_r R_2$)*
$\forall R_1 \in \mathbb{R}, R_2 \in \mathbb{R} : [\![R_1]\!] \subseteq [\![R_2]\!] \iff R_1 \leq_r R_2$.

Now, we state the complexity of the decision rules. First, auxiliary functions $\mathsf{pruneHead}(a, R)$ and $\mathsf{pruneNonStar}(R)$ can be computed in $O(|R|)$ time by choosing a suitable set data structure whose membership and union operations take $O(1)$ and $O(n)$ time, respectively, when n is the size of the set. Consider a relation $\{A\} \leq_r R$. If $|A| \leq 1$, then we can decide it in $O(1)$ time (by INCLEMPTYSET or INCLEMPTY). Otherwise, $|A| \geq 2$ and we use the rule INCLAREG or INCLSTARREG. The maximum size of the set obtained by repeatedly applying pruneHead and pruneNonStar to R is bounded by $|R| \cdot \max_{A' \in R} |A'|$ where $\max_{A' \in R} |A'|$ chooses the maximum of the lengths of atomic regular expressions in R. Since the size of A decreases by 1 in the rules INCLAREG and INCLSTARREG and both pruneHead and pruneNonStar functions can be computed in $O(|R| \cdot \max_{A' \in R} |A'|)$ time, we can decide a relation $\{A\} \leq_r R$ in $O(|A| \cdot |R| \cdot \max_{A' \in R} |A'|)$ time. Therefore, we can decide a relation $R_1 \leq_r R_2$ in $O(|R_1| \cdot \max_{A \in R_1} |A| \cdot |R_2| \cdot \max_{A' \in R_2} |A'|)$ time in the worst case. More detailed and formal analysis for the time complexity is available in our companion report [12].

6. Evaluation

In this section, we evaluate our new domain with its implementation in terms of analysis scalability and precision.

6.1 Methodology

Implementation. The prototype implementation of our domain extends SAFE [19, 20], which is a flow-sensitive and context-sensitive static analyzer for JavaScript web applications that give approximate heap states at each program point as analysis results. While SAFE supported only a variant of a constant string domain at first with two special elements, `NumStr` and `OtherStr`, that abstract all number strings and all non-number strings, respectively, it has been extended later to support a set domain whose elements are $\top_s$ (any string), $\bot_s$ (no string), `NumStr`, `OtherStr`, and a set with the maximum k number of concrete strings. Note that the formalization of the regular expression domain in Section 4 does not contain `NumStr` and `OtherStr` but the implementation incorporates them, which can be also incorporated in the formalization in a similar way that $\star$ is added in the domain.

While the regular expression domain can represent all elements in the set domain of SAFE so that it does not require any modification in the existing abstract semantics of string operations, we refined 5 operations to utilize the enhanced expressivity of the new domain elements: string concatenation, object property read, object property write, string conversion from an object, and string conversion from a function. We refined the first three operations as we described in Section 4. The last two operations return string values resulted from string conversion of an object and a function, respectively. According to the ECMAScript specification [1], the conversion for an object returns "`[object classname]`" where `classname` is the internal class name of the object, and the conversion for a function returns a source string for the function that starts with "`function`" followed by an implementation-dependent string. Since SAFE does not fully support internal class names of objects and implementation-dependent features, it conservatively models them as $\top_s$ in the set domain. However, we refined them as "`[object ⋆]`" and "`function⋆`" in the new domain. Note that the refinements improve the precision of the existing abstract semantics while preserving its soundness. Other string operations can be refined further to achieve even more improvements but we found that the refinements in the 5 operations already show significant improvements in analysis scalability and precision for SAFE. Throughout this section, we denote by SAFE_{reg} the implementation with the new domain and distinguish it from the original SAFE. We make the implementation open to the public [12].

Evaluation Methods. For evaluation target programs, we use a program group BENCH that includes 61 JavaScript programs from a jQuery tutorial[2]. The literature [3, 19] also used BENCH to evaluate JavaScript static analyzers. However, while all 61 programs in the previous work load a modified version of jQuery 1.10.0 where some statically indeterminate values are replaced with constant values, our target programs all use non-modified jQuery 1.7.0 because it is the highest version that both SAFE and TAJS can successfully finish analysis within the timeout of 10 minutes.

For analysis scalability, we compare the analysis results of our target programs with three static analyzers, SAFE, SAFE_{reg}, and TAJS. We refer interested readers to Section 7 for detailed explanation about SAFE and TAJS. As in the evaluation of previous work [3, 19], we run three analyzers with the target programs in the timeout of 10 minutes and count the number of programs whose analyses successfully complete within the timeout with normal exit heaps. In addition, we measure analysis time in analyzing 10 sample programs in BENCH to check how the new domain affects the analysis performance of SAFE in detail.

For analysis precision, we compare SAFE and SAFE_{reg} with the analysis results of 10 sample programs. We exclude TAJS for the comparison of analysis precision because fair measurement of analysis precision requires knowledge of analysis internals. We use the following three criteria that a previous work [20] used to measure analysis precision:

[2] `http://www.jquery-tutorial.net`

Group (# of programs)	Success (#)		
	SAFE	SAFE_{reg}	TAJS
BENCH (61)	27	40	2

Table 1: Numbers of successful analysis results in 10 min.

- MD : the number of object dereference points with multiple object values
- MC : the number of calls with multiple function values
- PR : the number of object property accesses with a property name approximated as non-constant

Lower MD, MC, and PR mean more precise analysis results. For example, for a method invocation `obj[x]()`, if the analysis result of the object dereference point `obj` contains multiple objects, MD includes it; if the analysis result of the method `obj[x]` contains multiple functions, MC includes it; if the analysis result of the object property name `x` is not a constant, PR includes it.

We performed all experiments on a Linux x64 machine with 3.4GHz Intel Core i7 CPU and 32GB memory, and all figures in the results are averaged over those from 5 runs.

6.2 Experimental Results

Scalability. Table 1 shows how many programs among 61 target programs in the BENCH group SAFE, SAFE_{reg}, and TAJS successfully finish analysis within the timeout of 10 minutes. While SAFE and TAJS complete the analysis of 27 and 2 programs, respectively, SAFE_{reg} performs the best with the complete analysis results of 40 programs. Table 2 shows detailed analysis results of 10 sample programs in the BENCH group. The first 5 programs are those that both SAFE and SAFE_{reg} successfully analyze within the timeout and the last 5 are those that only SAFE_{reg} successfully analyzes. The second and sixth columns of the table show analysis time that SAFE and SAFE_{reg} took to analyze each target program. We denote the timeout result by ✗. On average, the new domain reduced the analysis time of SAFE from 101.86 to 65.05 seconds for the first 5 programs and from minimum 10 minutes to 198.75 seconds for the rest 5 programs. Note that 4.html is the motivating example that we presented in Section 2 and we described that its analysis did not complete within 10 hours in SAFE. The regular expression domain can cause performance overhead as we discuss in the next section. However, this result demonstrates that the new domain significantly improves the scalability of an existing analyzer in analyzing jQuery programs rather than causing extra performance overhead.

Precision. Table 2 shows analysis precision of SAFE and SAFE_{reg} in analyzing 10 sample programs. For the first 5 programs that SAFE successfully analyzes within the timeout, SAFE_{reg} shows relatively small improvement compared to the results for the rest 5 programs because SAFE already shows reasonable analysis precision. While MC remains the same in both analyzers, MD and PR decreased from 21 and 13 to 12 and 5, respectively, on average in SAFE_{reg}. On the other hand, for the rest 5 programs that only SAFE_{reg} successfully analyzes within the timeout, SAFE_{reg} shows great improvement. On average, it reduces MD, MC, and PR from 218, 50, and 40 to 23, 3, and 16, respectively. This result demonstrates not only that improving analysis precision also improves analysis scalability but also that the regular expression domain significantly improves analysis precision of an existing analyzer. Note that refining more string operations may further enhance the improvement in analysis scalability and precision shown in this section.

6.3 Discussion

Limitations and threats to validity. Analysis results in the regular expression domain are always more precise than or as precise as those in constant and set string domains since the domain includes all elements in the other two domains with additional regular expression elements that refine the $\top_s$ element of the other two domains. However, generally speaking, the domain can cause extra performance overhead; while the decision complexity for order relations between elements in constant and set domains is constant and linear, respectively, the complexity for the regular expression domain is biquadratic depending on the number and the maximum length of atomic regular expressions. But because more imprecise results can incur more performance overhead, if the performance benefits from improved precision are bigger than the performance overhead by the order relation decision, the new domain leads to faster analysis than the other two. We demonstrated such cases in the previous section.

A possible threat to validity of our results is that we used simple programs that use jQuery on the experiment and thus conclusions drawn from the results may not hold in real-world applications. However, even state-of-the-art static analyzers such as SAFE and TAJS are not able to analyze such simple programs because of the dynamic natures of jQuery. We believe that making existing analysis scalable enough to analyze simple programs is an essential step towards scalable analysis of real-world applications, and our technique contributes to it. Another threat is that analysis results regarded as successful in SAFE and TAJS may not be sound because of possible unsoundness in the implementation of SAFE and TAJS. Although SAFE and TAJS provide comprehensive models of browser environments [10, 20], analysis of programs that use unmodeled features may yield unsound results. However, since it is orthogonal to the technique presented in this paper, we assume that the base analyzers provide sound analysis results for target programs.

Other causes for scalability problems. We investigated why SAFE_{reg} still could not complete the analysis of some programs within 10 minutes and observed two reasons: 1) a sound event system in SAFE and 2) statically indeterminate values in AJAX communication. The SAFE event

Target Program	SAFE				$SAFE_{reg}$			
	Time (s)	MD (#)	MC (#)	PR (#)	Time (s)	MD (#)	MC (#)	PR (#)
1.html	120.07	24	3	12	82.72	16	3	5
10.html	112.90	24	3	12	75.73	16	3	5
20.html	93.72	20	3	12	57.04	9	3	5
30.html	93.86	18	3	12	57.54	9	3	5
54.html	88.75	17	3	16	52.23	9	3	6
Average	101.86	21	3	13	65.05	12	3	5
4.html	✗	321	159	41	483.60	16	3	8
29.html	✗	208	55	29	113.83	23	3	10
35.html	✗	200	32	42	99.58	26	3	22
57.html	✗	180	3	43	147.93	25	3	20
60.html	✗	181	3	43	148.84	25	3	20
Average	✗	218	50	40	198.75	23	3	16

Table 2: Analysis time and precision of SAFE and $SAFE_{reg}$ for 10 jQuery tutorial programs with the timeout of 10 minutes.

system that approximates all possible execution sequences of registered event handlers and AJAX communication that receives arbitrary values produce over-approximate values, which cause scalability problems. We believe that more elaborate event analysis systems and developers' hints for values from AJAX communication would alleviate the problems.

Analysis of non-jQuery programs. We further evaluated our regular expression domain with 21 non-jQuery benchmarks from a previous work [13]; the benchmarks consist of 7 benchmarks from SunSpider and V8, 7 Firefox browser addons, and 7 programs from open source JavaScript frameworks and their test suites[3]. While the regular expression domain can cause performance overhead, we found that it only causes negligible overhead in analyzing the target benchmarks. Both SAFE and $SAFE_{reg}$ could not complete 3 programs within the timeout of 10 minutes, but the average analysis time over the results finished in 10 minutes were 49.02 (min.: 0.64, max.: 575.58, median: 4.04, standard deviation: 136.20) and 49.45 (min.: 0.63, max.:576.67, median: 4.09, standard deviation: 136.37) seconds in SAFE and $SAFE_{reg}$, respectively. As for precision, the analysis results from all targets showed the same MD, MC, and PR in SAFE and $SAFE_{reg}$. We believe that this is because we refined only 5 string operations where only 3 operations introduce regular expression elements and the targets do not use them much.

We found that evaluating our domain with real-world applications such as web pages is premature because analysis of such applications involves many challenges such as AJAX communication and heavy user interaction by events, which are beyond the scope of this paper. Although static analysis for JavaScript does not scale to real-world applications yet, we believe that our domain itself in $SAFE_{reg}$ would not cause much performance overhead compared to constant and set domains in deciding order relations between elements. As we discussed before, the complexity of the order decision depends on the number and maximum length of atomic regular expressions, which we expect to be small numbers due to only 3 operations that introduce simple regular expressions. However, with more operations that can introduce complex regular expressions, the domain can cause more overhead, where it is possible to restrict such numbers as some maximum values to minimize the complexity in the domain.

[3] `http://www.defensivejs.com`, `http://linqjs.codeplex.com`

7. Related Work

JavaScript static analysis frameworks. SAFE [14, 19], JSAI [13], and TAJS [3, 9] are JavaScript static analyzers that produce approximate heap information at each program point. All of them faithfully model the ECMAScript semantics with some built-in function modeling. In addition, SAFE and TAJS provide a comprehensive modeling of browser environments [10, 20] to analyze web applications. While they use different lattice domains for abstract values, we formalized the conceptually common model of strings, objects, and main operations on them in Section 3.

SAFE and TAJS have made significant progress in analyzing jQuery. Loop-sensitive analysis (LSA) of SAFE [19] significantly improves the analysis precision and the scalability enough to analyze all released versions of jQuery within a practical time limit. We have applied the new domain to SAFE with LSA, and we showed that the domain further improves its precision and scalability. TAJS [3] can analyze 11 out of all 12 released jQuery 1.x versions within one minute. But the experiments used modified versions of jQuery where some statically indeterminate values like `Math.random()` were replaced with constant values. We found that TAJS can analyze 4 out of 12 jQuery 1.x versions within 10 minutes when it analyzes non-modified jQuery.

WALA [22, 23] supports static pointer analysis of Java and JavaScript programs. It can analyze 3 versions of jQuery. While we can apply our regular expression domain to WALA

as well, we did not consider WALA because SAFE and WALA provide different information as analysis results as heap approximation and pointer information, respectively.

Dynamic object and jQuery typing. Politz *et al.* [21] presented a type system for dynamic objects commonly used in dynamic languages such as JavaScript, Python, and Ruby to guarantee that well-typed programs do not read absent properties of objects during execution. Their object type is a map from property names to property types where the property names can be any patterns such as regular expressions unlike our object model that has concrete string names for properties. Their type system parameterizes the pattern domain with a restriction that the inclusion between patterns should be decidable. Our regular expression domain is an instance of the type system satisfying the restriction.

Lerner *et al.* [15] proposed a type system for jQuery considering it as a language. The type system has two novel concepts, *multiplicity* that represents the possible size of object collections and *local structure* that represents types only for local parts of a web document that the target jQuery program uses. The type system detects misuses of jQuery APIs such as operations on empty object collections. However, the type system requires user annotation for the local structure, which subsequently requires understanding of the interaction between web documents and jQuery programs. While our analysis does not require any user annotation, it has higher time complexity by fixpoint computation than the type system and it cannot handle events precisely.

String and regular expression domains. Madsen and Andreasen [16] suggested 12 string domains, among which 7 are new, for precise analysis of dynamic property accesses in JavaScript. They showed that a hybrid domain of string set, character inclusion, and string hash domains produced the best analysis precision and scalability. However, even the hybrid domain cannot represent string values resulted from concatenation of a constant string with multiple characters and a statically indeterminate value, which are frequently used as keys to dynamic property accesses in jQuery.

Researchers have used regular expressions for Java string analysis. Christensen *et al.* [5] presented a string analysis that produces, for each string operation in a program, a regular expression that approximates string values resulted from the operation. The analysis represents a program as a def-use graph of string data in string operations and constructs a multi-level finite automaton (MLFA) using the context-free and regular grammars translated from the graph. Multiple deterministic finite-state automata (DFA) can be derived for each string operation from MLFA and then the regular expression that each DFA represents is an analysis result for the result string values of the operation. The analysis does not consider inclusion relations between regular expressions during analysis because regular expressions appear only as analysis results. However, the complexity of extracting DFA is doubly exponential in the worst case while our domain requires the polynomial-time complexity. In addition, the analysis requires a def-use graph for a target program, which is still a research problem for JavaScript.

Choi *et al.* [4] formalized a string analysis using regular expressions in abstract interpretation like our approach. Their regular expression domain is more expressive than ours in that their domain allows the Kleene star operation (*) on arbitrary combinations of regular expressions while ours allows it only on disjunctions over all alphabets. They also presented complex widening rules to minimize precision loss by join operations on regular expressions. While we define the order relation between regular expressions using the inclusion relation between them, they do not present how to decide the relation. Note that, without the order relation decision, the analysis may perform unnecessary widening for regular expressions already in the order relation.

Regular expression inclusion. To use regular expressions in a sound way as elements of abstract string domains, one should define the order relation between regular expressions using the inclusion relation between them. If one regular expression R_1 *includes* another R_2, it means that a set of all strings that R_1 matches includes a set of all strings that R_2 matches. However, deciding inclusion for general regular expressions has been known as a problem in PSPACE-complete [24]. Fortunately, previous work [2, 18] has found tractable cases for some restricted subset of regular expressions and Martens *et al.* [17] studied the complexity of inclusion for various subsets of regular expressions including previously known tractable cases. According to the study, surprisingly, inclusion decision between elements in a simple domain whose element is a sequence of a and $(a_1|...|a_n)^*$ (a, a_i, and a_n are alphabets with $n \geq 1$ and $|$ and $*$ are usual disjunction and Kleene star operations) already reaches the maximum complexity, PSPACE-complete. To the best of our knowledge, the complexity of inclusion for our regular expressions is previously unknown, and we presented polynomial-time inclusion decision rules for them.

Other applications. Regular expressions are widely used in various applications including pattern matching with strings. Regular expressions for string matching are often called regexes and supported by many languages such as Perl, JavaScript, and Ruby. Regexes are used to detect SQL injection attacks[4] by checking if a user input contains SQL specific meta-characters and commands. Because most regexes implementations include non-regular features, they are more expressive than ours but with higher complexity.

Another well-studied example is XML type checking using regular expression types to describe XML types such as DTD and XML Schema. Regular expression types [7] consist of primitive types, pair types, recursive types, and union types, and they subsume our regular expressions. While we

[4] `http://www.symantec.com/connect/articles/detection-sql-injection-and-cross-site-scripting-attacks`

can use regular expression types, the complexity of their inclusion relation is much higher as EXPTIME-complete. Instead, we devised a subclass of regular expressions that is simple but powerful enough for our problem and whose inclusion relation can be efficiently decided in PTIME.

8. Conclusion

We proposed a novel abstract string domain whose elements are simple regular expressions for precise and scalable static analysis of jQuery, the most popular JavaScript library. The domain is expressive enough to represent prefix, infix, and postfix substrings of a string and even their sets. In a theoretical aspect, we formalized the domain and related operations in abstract interpretation with the soundness proofs. In a practical aspect, we formally presented polynomial-time inclusion decision rules between the regular expressions with the soundness and completeness theorems for the implementation of the domain. We have implemented the domain as an extension of SAFE and demonstrated that the technique significantly improves the scalability and precision of SAFE in analyzing 61 programs that use jQuery. We make the formal proofs and the implementation open to the public.

Acknowledgments

This work was supported in part by Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grants NRF-2014R1A2 A2A01003235 and 2016R1C1B1015095) and Samsung Electronics.

References

[1] ECMAScript Language Specification. Edition 5.1, 2011.

[2] P. A. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *Proceedings of the 10th International Conference on Computer Aided Verification*. Springer Berlin Heidelberg, 1998.

[3] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, 2014.

[4] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A practical string analyzer by the widening approach. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, 2006.

[5] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the International Symposium on Static Analysis*, 2003.

[6] 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 the Symposium on Principles of Programming Languages*, 1977.

[7] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2005.

[8] D. Hovland. The inclusion problem for regular expressions. *Journal of Computer and System Sciences*, 78(6), 2012.

[9] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proceedings of the International Symposium on Static Analysis*. Springer-Verlag, 2009.

[10] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the International Symposium on Foundations of Software Engineering*. ACM, 2011.

[11] jQuery Foundation. jQuery. `http://jquery.com`.

[12] KAIST PLRG. `http://plrg.kaist.ac.kr/pch`.

[13] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the International Symposium on Foundations of Software Engineering*, 2014.

[14] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *Workshop on Foundations of Object Oriented Languages*, 2012.

[15] B. S. Lerner, L. Elberty, J. Li, and S. Krishnamurthi. Combining form and function: Static types for jQuery programs. In *Proceedings of the European Conference on Object-Oriented Programming*, 2013.

[16] M. Madsen and E. Andreasen. String analysis for dynamic field access. In *Proceedings of the International Conference on Compiler Construction*, 2014.

[17] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*, 2004.

[18] T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of the International Conference on Database Theory*, 1999.

[19] C. Park and S. Ryu. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *Proceedings of the European Conference on Object-Oriented Programming*, 2015.

[20] C. Park, S. Won, J. Jin, and S. Ryu. Static analysis of JavaScript web applications in the wild via practical DOM. In *Proceedings of the International Conference on Automated Software Engineering*, 2015.

[21] J. G. Politz, A. Guha, and S. Krishnamurthi. Semantics and types for objects with first-class member names. In *Workshop on Foundations of Object Oriented Languages*, 2012.

[22] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic determinacy analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2013.

[23] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming*, 2012.

[24] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time(preliminary report). In *Proceedings of the Annual ACM Symposium on Theory of Computing*, 1973.