String Analysis for Dynamic Field Access

Magnus Madsen and Esben Andreasen

Aarhus University
{magnusm,esbena}@cs.au.dk
http://cs.au.dk/~{magnusm,esbena}

Abstract. In JavaScript, and scripting languages in general, dynamic field access is a commonly used feature. Unfortunately, current static analysis tools either completely ignore dynamic field access or use overly conservative approximations that lead to poor precision and scalability.

We present new string domains to reason about dynamic field access in a static analysis tool. A key feature of the domains is that the equal, concatenate and join operations take $\mathcal{O}(1)$ time.

Experimental evaluation on four common JavaScript libraries, including jQuery and Prototype, shows that traditional string domains are insufficient. For instance, the commonly used constant string domain can only ensure that at most 21% dynamic field accesses are without false positives. In contrast, our string domain $\mathcal{H}$ ensures no false positives for up to 90% of all dynamic field accesses.

We demonstrate that a dataflow analysis equipped with the $\mathcal{H}$ domain gains significant precision resulting in an analysis speedup of more than 1.5x for 7 out of 10 benchmark programs.

1 Introduction

JavaScript is a notoriously difficult language for static analysis due to its many dynamic features, including a flexible object-model, prototype-based inheritance, dynamic property accesses[1], non-standard scope rules, coercions, and the `eval`-construct [3, 5, 6, 9, 10, 13].

This paper focuses on the problem of dynamic property accesses in points-to or dataflow analysis of JavaScript, that is, reads or writes to objects where the property names are computed on-the-fly. This involves statements such as `v = o[p]` or `o[p] = v` where the value of `p` is not statically known. A simple sound approach is to treat the first statement as a read of *any* property of `o` and the second statement as a write to *all* properties of `o`. However, such an approach loses the benefits of field-sensitivity. And, as the following sections illustrate, it is too imprecise in practice. In JavaScript, string manipulations and dynamic property accesses are common, and to paraphrase an old mantra: *"One man's string is another man's heap location".*

[1] In JavaScript a field is called a property and reading/writing a field is called a property access. We will use this terminology for the remainder of the paper.

Dynamic Reads The JavaScript code below shows three different ways of accessing a property of an object `o`.

```
1  x = o.p;                // a static read of 'p'
2  x = o["p" + "q"];       // a dynamic read of 'pq'
3  x = o[c ? "p" : "q"]; // a dynamic read of 'p' or 'q'
```

Line 1 is straightforward to analyze. Line 2 can be handled using syntactic constant folding. However, if the concatenation involves variables or heap locations the syntactic approach is no longer viable, instead some kind of string analysis is required. Line 3 is even more nefarious for a static analysis. If the statement is analyzed using the constant string lattice – without context sensitivity or path sensitivity – the result will be $\top$ (corresponding to any property) and thus it is unknown which property is read from `o`. A sound analysis will then conservatively include *all* properties accessible on the `o` object in the result. However this includes all properties available in the prototype hierarchy of `o`! If `o` is a regular object and its prototype is `Object[[proto]]` then around 10 properties are involved, including functions such as `toString` and `__defineSetter__`. If the internal prototype object is `Array[[proto]]` then the problem is exacerbated by an *additional* 20 properties, including mutators such as `pop`, `push`, and `reverse`, leading to even more spurious flow.

Dynamic Writes The JavaScript code below shows three different ways to store a value into an object property.

```
1  o.p = function() {}
2  o.["p" + "q"] = function() {}
3  o.[c ? "p" : "q"] = function() {}
```

The first two statements can be handled like in the previous section. However, the third statement requires extra care. If it is not known to which property a value is written, then the analysis must conservatively write it to *all* properties of that object using a weak update, i.e. by joining the new value into the existing values. Thus, after the last statement, any property of object `o` can point to the function defined on line 3.

Dynamic Reads and Writes Even more precision is lost when dynamic reads and writes are combined as shown below:

```
o[p][q] = function() {};
```

If neither `p` nor `q` are known by the analysis, e.g. if the constant string lattice is $\top$ for both, then `p` could potentially be the string "`__proto__`" and as a result `o[p]` could be the internal prototype object of `o`. If `o` is a regular object then this would be the `Object[[proto]]` object. Thus, the write will cause the function to be written to *all* properties of the `Object[[proto]]` object which is shared by *all* JavaScript objects. In Java, for instance, this would correspond to overriding all fields and methods of the `java.lang.Object` with a spurious

function. To handle such scenarios, a better string abstraction is required, in particular, the abstraction of `p` and `q` should be able to rule out property names such as `__proto__`. Furthermore, should a loss of precision occur for `p`, then the abstraction of `q` should still limit the damage done to `Object[[proto]]` by writing to just a few of its properties.

Event Handlers An additional challenge occurs for JavaScript web applications. In JavaScript, an event handler may be registered on a HTML object by writing to several special properties, e.g. `onclick`, `ondblclick`, `onload` and `onsubmit`. For instance, writing a function value to the `onclick` property registers that function as a callback which is executed whenever the user clicks the mouse on its corresponding HTML object.

A sound analysis must take such registrations into account. If a dynamic property write occurs, where a HTML object is the base object, and the analysis cannot rule out that the write occurs to one of these special properties, then it must conservatively assume that an event handler registration occurs. This can lead to spurious event handler registration and spurious dataflow.

Usage in Practice According to a study of JavaScript behavior by Richards et al. [14]: 8.3% of all property reads are dynamic and 10.3% of all property writes are dynamic (c.f. Section 5.2 in [14] and the associated web page[2]). Furthermore, as Table 5 shows, many popular JavaScript libraries contain several hundred dynamic property reads and writes.

Contributions In summary our paper makes the following contributions:

- We describe twelve different string abstractions – five previously known and seven new. We focus on abstractions which require $\mathcal{O}(1)$ space and support the equal, concatenate and join operations in $\mathcal{O}(1)$ time. We place a strong emphasis on the precision and performance of the equal operation.
- We experimentally evaluate each string abstraction on four common JavaScript libraries: jQuery, Prototype, MooTools and jQuery UI. We base our evaluation on concrete executions of each library thus providing an analysis independent upper bound on the precision of each string abstraction.
- We propose a precise and efficient string abstraction $\mathcal{H}$ for reasoning about dynamic property accesses. Experiments show that $\mathcal{H}$ has no spurious flow for up to 90% of all dynamic property accesses compared to at most 21% for the constant string abstraction.
- We equip a dataflow analysis with the proposed $\mathcal{H}$ string abstraction and show that it leads to a significant improvement in precision and performance. In particular, the analysis achieves a speedup of at least 1.5x for 7 out of 10 benchmark programs.

[2] http://dumbo.cs.purdue.edu/js/analysis-charts/events.html

2 Related Work

We begin with a discussion of prior work related to string analysis and JavaScript.

String Analysis Costantini et al. presents an abstract interpretation-based framework for string analysis and instantiates the framework for four different abstract domains [4]: a) The *character inclusion* domain, which tracks what characters *may* or *must* occur within the string, b) the *prefix/suffix* domain, which tracks the k first and last characters of the string, c) the *bricks* domain, where a brick $b = [\mathcal{P}(s)]_{min}^{max}$ represents all strings that can be generated by concatenating elements of $\mathcal{P}(s)$ between min and max times, and d) the *string graph* domain for which we refer the reader to [4] for details. Costantini et al.'s work does not discuss string equality which is a key issue for our work. Another difference is that Costantini et al. focus on the theoretical aspects of the strings domains, whereas we provide an experimental evaluation of the precision and performance of the domains.

Christensen et al. presents the Java String Analyzer [2] (JSA), a static analysis tool which approximates string expressions in a Java program by a regular language. The technique is based on translation from the control-flow graph into the def-use graph, which is then translated to a context-free grammar and finally widened into a regular language. JSA has found a wide variety of applications; including verification of generated SQL statements and validation of dynamically constructed HTML. In comparison to our work Christensen et al. focus on string analysis in general, whereas we focus on string analysis for reasoning about dynamic property accesses. Furthermore, we place a strong emphasis constant time and space bounds for our abstract domains compared to the potentially exponential time bound for the whole JSA analysis.

Zheng et al. present Z3-str a general purpose string solver based on the Microsoft Z3 SMT solver [16]. The solver models strings as a primitive type together with booleans and integers. Its supported operations include concatenation, equality, sub-string and replace. Kiezun et al. present HAMPI a string solver based on constraints on regular languages and fixed-size context-free languages [11]. In relation to our work, general purpose string solvers such as Z3-str and HAMPI, are heavy-weight. We aim to construct a light-weight string domain, which can be used in any points-to or dataflow analysis, to address the problem of dynamic property accesses.

JavaScript Analysis Guarnieri et al. present GATEKEEPER, a tool for static enforcement of security policies for JavaScript programs [6]. The authors present an Andersen-style [1] inclusion-based, context-insensitive, points-to analysis for JavaScript. GATEKEEPER classifies whether JavaScript "widgets" are safe with respect to a security policy by inspecting information from the computed points-to sets and call graph. GATEKEEPER cannot soundly reason about dynamic property accesses and thus must resort to runtime enforcement of the security policy for every dynamic read or write (c.f. Section 3.2.2, [6]).

Guarnieri et al. present ACTARUS, a static taint analysis for JavaScript [7]. ACTARUS tracks information flow to ensure that data from an untrusted source cannot reach a high-integrity sink. The analysis, like the GATEKEEPER project, is based on inclusion-based points-to analysis. ACTARUS handles dynamic property accesses (called reflective property accesses in their paper) by keeping known string constants separated and creating new abstract objects when strings objects are concatenated (Section 3.3 in [7]). Yet, abstraction must be introduced at some point, and it is not clear from the paper, how this is implemented in ACTARUS.

Jensen et al. present the Type Analysis for JavaScript (TAJS) tool based on inter-procedural dataflow analysis [10]. The analysis aims for soundness and goes to great lengths to faithfully model the semantics of JavaScript. The string abstraction is based on the constant string lattice extended to track whether the string may or must be a number-string. In more recent work, Jensen et al. extends TAJS with the *Unevalizer*, a technique for analyzing certain invocations of `eval` [8]. For this purpose, the string lattice is extended to track strings which are valid JavaScript identifiers or contain characters which are valid inside identifiers. Jensen et al. originally identified the problem of dynamic property writes to HTML objects [9].

Sridharan et al. present *correlation tracking*, a technique for identifying and tracking dynamic property reads and writes which are related [15]. The purpose of their technique is to ensure that e.g. for-each-in loops which copy properties from one object $\mathtt{o}_{src}$ to another $\mathtt{o}_{dst}$ maintain the relation s.t. $\mathtt{o}_{dst}[p] = \mathtt{o}_{src}[p]$. Thus, preserving field-sensitive precision. We believe that correlation tracking is a step in the right direction for scaling points-to and dataflow analyses for large JavaScript libraries. However, not all dynamic property accesses are correlated and this paper presents an orthogonal way to improve precision.

In summary, except for Sridharan et al., most work use very simple techniques for dealing with dynamic property accesses.

3 String Domains

In this section we present some existing and several new abstract string domains. We have marked the domains which we believe are new to the literature with the $\star$ symbol.

Assumptions We assume, for the rest of the paper, an underlying points-to or dataflow analysis with a standard field-sensitive heap abstraction. It is our goal to design string lattices which can be used together with the analysis without increasing its running time.

String Operations JavaScript has around 15 built-in string operations. We consider the abstract equality ($\hat{=}$), abstract concatenation ($+$) and lattice join ($\sqcup$) operations central for reasoning precisely and efficiently about dynamic property accesses. The $\hat{=}$ operation is applied at every dynamic property access to

decide which property names may be referenced. Thus, it must be both *precise* – to rule out many property names – and *efficient* since it will be evaluated often. Similarly, the $+$ and $\sqcup$ operations should be efficient, while maintaining as much knowledge about the underlying strings as possible. Additional string operations are discussed in Section 3.16. All domains described in the following have finite height, thus widening is not required to ensure termination.

3.1 Constant String

The *constant string* lattice $\mathcal{C}$ tracks a single concrete string. The lattice is elements are $\bot, \top$ and $s \in Str$ where $\bot$ and $\top$ are the bottom and top elements, respectively. The $\bot$ element represents no concrete strings, whereas $\top$ represents all possible concrete strings. The lattice supports the equal, concatenate and join operations in $\mathcal{O}(n)$ time in the length of the string. In practice most strings are short so we do not consider the linear complexity to be a problem. The constant string lattice is the standard solution used by much prior work (as discussed in Section 2) and is used as the baseline abstraction in Section 4.2.

3.2 String Set

The *string set* lattice $\mathcal{SS}$ is the powerset lattice ordered by subset-inclusion of a bounded number of concrete strings. The lattice elements are $\top$ and $\{s | s \in \mathcal{P}(Str) \wedge |s| \leq k\}$ where s is a set of up to k strings and $\top$ represents all possible concrete strings. The lattice supports the equal, concatenate and join operations in $\mathcal{O}(k^2 \times n)$ time, where k is the bound and n is length of the longest string.

3.3 Length Interval

The *length interval* lattice $\mathcal{I}$ is the interval lattice on the string length. It tracks the minimum and maximum length of the concrete strings it represents. The length interval lattice can distinguish property names which are usually short, from data strings such as HTML code, image data or other serialized data. The interval representation is standard, with a bounded width k, and supports the equal, concatenate and join operations in $\mathcal{O}(1)$ time. Finally, we note that the length interval lattice can be useful for coercions from strings to booleans as it tells us whether the string may be the empty string, and thus can coerce to `false`.

3.4 Length Hash ⋆

The length interval lattice $\mathcal{I}$ loses much precision whenever strings of disparate length are joined. We propose to overcome this by introducing the length hash lattice $\mathcal{LH}$. The length hash lattice tracks *a set of string length hashes* instead of tracking the minimum and maximum string length. We take a universe of fixed size $U = \{0 \ldots b\}$ and a hash function $h : S \to U$ s.t. each string length

```
def concat(A: Long, B: Long): Long = {
  var R: Long = Long.reverse(B);
  var C: Long = 0L;
  for (i <- 0 until b) {
      r = Long.rotateLeft(r, 1);
      if ((A & R) != 0L) {
        C |= (1 << i);
      }
  }
  return C;
}
```

Fig. 1: Implementation of fast hash concatenation in Scala. In Java/Scala bit positions are indexed in the opposite direction of what we have described on thus `rotateLeft` is used instead of a right rotate.

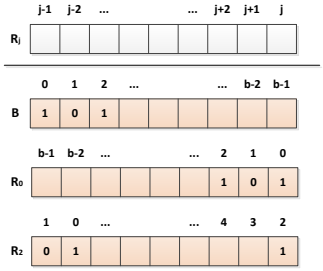

Fig. 2: The top part of the figure shows the bitvector R_j, obtained by reversing and right-rotating B j times. The bottom part is an example where R_0 and R_2 are obtained from the bitvector B.

hashes to a particular bucket in the universe. The lattice is the powerset lattice of U ordered by subset-inclusion (i.e. $\bot$ is the empty set and represents no concrete strings). If we fix b at the word size of the target architecture we can efficiently implement $\mathcal{LH}$ as a bitset. The equal and join operations can then be implemented as bitwise operations in $\mathcal{O}(1)$ time.

Concatenation is more tricky. If we require the hash function h to be distributive, s.t. $(h(s_1 + s_2) = h(s_1) + h(s_2) \mod b)$, then concatenation can be implemented precisely. Concatenation of the abstract strings $\hat{s}_1$ and $\hat{s}_2$ is computed by summing all lengths in $\hat{s}_1$ with all lengths in $\hat{s}_2$ and taking the modulus. A naive implementation calculates these sums inside two nested loops. The complexity of this implementation is $\mathcal{O}(b^2)$ where b is the size of the universe. This is $\mathcal{O}(1)$ since b is a fixed constant, but in practice $b = 64$ and thus the naive implementation may require up to 4096 iterations.

A better solution achieves $\mathcal{O}(b)$ time by only iterating through the lengths of $\hat{s}_1$ and summing with the lengths of $\hat{s}_2$ *simultaneously* by using a few clever bit operations. Let A and B be the bitvectors representing $\hat{s}_1$ and $\hat{s}_2$ respectively. We observe that the k'th position in the resulting bitvector C depends on all $A[i]$ and $B[j]$ where $i + j \equiv k \mod b$.

We define R_j to be the bitvector obtained from B by first reversing it and then right rotating the result j positions. Thus, e.g. R_0 is the reverse of B and R_2 is the reverse of B right rotated two positions, as shown in Figure 2.

We can now compute $C[k]$ by evaluating $A \wedge R_{k+1} \neq 0$, since $R_{k+1}[i] = B[(b-1-i)+(k+1) \mod b] = B[k-i \mod b] = B[j]$ and thus:

$$C[k] = (A \wedge R_{k+1} \neq 0) = \bigvee_{i=0}^{b-1} A[i] \wedge B[j]$$

which is equivalent to what is computed by the naive implementation. The code in Figure 1 implements this strategy. In a synthetic benchmark the above code resulted in a factor 70 speedup compared to the naive implementation.

As an example, the abstraction of {`abc`, `abcdef`} is a bitset containing the elements 3 and 6. This bitset represents all strings of length $\{l | l = 3 + b * i \vee l = 6 + b * i, \forall i \geq 0\}$.

3.5 Prefix and Suffix Characters

The *prefix-suffix character* lattice $\mathcal{PS}$ tracks the first and last character symbol of the string. It is formed as the cartesian product of two constant character lattices; one for the prefix and one for the suffix. The lattice supports the equal, concatenation and join operations in $\mathcal{O}(1)$ time.

In jQuery HTML tags can be passed into to the `$`-function to construct new HTML elements. Inside the `$`-function[3], the following test is used to inspect whether an argument is an HTML tag:

```
var length = selector.length;
if (selector.charAt(0) === "<" &&
    selector.charAt(length - 1) === ">" &&
    length >= 3) {
```

The prefix-suffix character lattice can analyze code like the above by providing information about whether the first and last character *may* or *must not* be the `<` and `>` characters, respectively.

3.6 Character Inclusion

The *character inclusion* lattice $\mathcal{CI}$ tracks what character symbols *may* and *must* occur within a string. It is formed as the cartesian product of the four sub-lattices: c_{may}, c_{must}, e_{may} and e_{must}. The c_{may} and c_{must} lattices are powerset lattices of character symbols ordered by subset- and superset inclusion, respectively. The e_{may} and e_{must} boolean lattices tracks whether the concrete set of strings may or must include the empty string or a character symbol which is not representable by c_{may} or c_{must}. As an example, the empty string, and the strings `foo` and `moo` are represented as:

$$\mathcal{CI} = (c_{\text{may}} = \{\texttt{f}, \texttt{m}, \texttt{o}\}, c_{\text{must}} = \{\texttt{o}\}, \top_{e_{\text{may}}}, \bot_{e_{\text{must}}})$$

The equal operation of $\mathcal{CI}_1$ and $\mathcal{CI}_2$ is implemented as:

1. If $\mathcal{CI}_1$ or $\mathcal{CI}_2$ is $\bot_{\mathcal{CI}}$ then the result is $\bot_{\text{bool}}$, i.e. if one (or both) of the lattices represents the empty set of concrete strings then the results represents the empty set of concrete booleans (denoted by $\bot_{\text{bool}}$).
2. If $c^1_{\text{must}} \cap c^2_{\text{may}} = \emptyset$ or $c^2_{\text{must}} \cap c^1_{\text{may}} = \emptyset$ the result is `False`, i.e. if a character *must* be in $\mathcal{CI}_1$ but at the same time is *definitely not* present in $\mathcal{CI}_2$ the strings cannot be the same (and vice versa).
3. If $c^1_{\text{may}} \cap c^2_{\text{may}} = \emptyset$ and $e^1_{\text{may}} = e^2_{\text{may}} = \bot$ then the result is `False`, since no characters overlap between $\mathcal{CI}_1$ and $\mathcal{CI}_2$, and none of them are the empty string.

[3] jQuery v. 1.8.3, line 114

4. If $\mathcal{CI}_1$ must contain the empty string or an unrepresented character and $\mathcal{CI}_2$ definitely does not (or vice versa) the result is `False`, since either contains characters which the other does not.
5. Otherwise the result is $\top_{\text{bool}}$, i.e. the concrete set of `true` and `false`.

We implement the character inclusion lattice using two bitsets. The first bitset tracks may-information and the second tracks must-information. In each bitset we use a bit to track whether the string may/must be the empty string or contain an unrepresentable character. The remaining bits are reserved for character symbols. If we use a word size of 64 this leaves space for 63 character symbols.

We represent character symbols in the ASCII range from 32 to 95, which includes the characters 0-9, A-Z, the special characters `!"#$%&'()*+,-./:;<=>?@` and space. Lowercase letters can be accommodated by converting them to uppercase, i.e. the character inclusion lattice is case-insensitive. In summary, the bitset-based character inclusion lattice supports the equal, concatenate and join operations in $\mathcal{O}(1)$ time.

3.7 Index Predicate ⋆

The *index predicate* lattice ($\mathcal{IP}$) tracks whether a boolean valued predicate $\rho(c)$ may or must hold for the character symbol c at index i of the string, where the index is bounded by a constant b. That is, the lattice only tracks the predicate for the first b characters. Most property names are short, and thus having incomplete information for long strings is unlikely to be a problem in practice. We can instantiate the lattice with predicates like the following:

- *Lowercase / Uppercase* – whether the character at index i may or must be a lowercase or uppercase letter. This is useful for property names that use camel casing, e.g. `hasOwnProperty`.
- *Underscore* – whether the character at index i may or must be an underscore. Like above, this is useful for "hidden" property names with underscores, e.g. `__defineGetter__`.
- *Digit* – whether the character at index i may or must be a digit. If all character symbols must be digits then the entire string represents a number.
- *Non-identifier Character* – whether the character at index i may or must be a non-identifier character. (A generalization of the `idPart` in *Unevalizer* [8])
- *Whitespace* – whether the character at index i may or must be white space (i.e. space, tabs or newline) which is useful for e.g. `split`.

The index predicate lattice is the cartesian product of two powerset lattices of indices $i_{\texttt{may}}$ and $i_{\texttt{must}}$ and the length interval lattice. The length interval lattice is used to handle concatenation precisely.

We implement the $i_{\texttt{may}}$ and $i_{\texttt{must}}$ lattices as bitsets for the first 64 string indices. The length interval lattice uses the standard representation. The join operation is straightforward to implement in $\mathcal{O}(1)$ time. The equal operation can be implemented similarly to the equal operation for the character inclusion

lattice. The concatenate operation, however, requires more legwork. If the strings $s_1 = (i^1_{\texttt{may}}, i^1_{\texttt{must}})$ and $s_2 = (i^2_{\texttt{may}}, i^2_{\texttt{must}})$ are concatenated and the length of s_1 is not an interval, but a concrete number, then concatenation is simply a matter of merging the $i^1_{\texttt{may}}$ and $i^2_{\texttt{may}}$ bitsets using the concrete length of s_1 as an offset. On the other hand, if the length of the string s_1 is an interval then the $i^1_{\texttt{may}}$ and $i^2_{\texttt{may}}$ bitsets must be merged by all offsets in that interval. Similarly for the must bitsets, as shown in Figure 3.

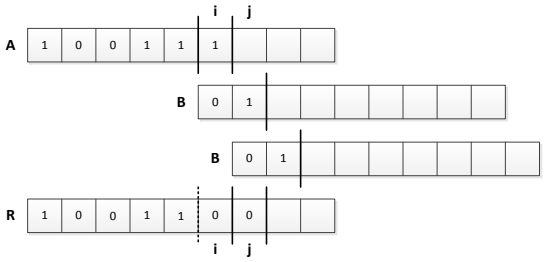

Fig. 3: Concatenation of two index predicate lattices `A` and `B` for the $i^1_{\texttt{must}}$ and $i^2_{\texttt{must}}$ sets, respectively. Here the length of `A` is between $[5, 6]$. The example shows how the indices `i` and `j` are computed by bitwise-and.

3.8 Sliding Index Predicate ⋆

The *sliding index predicate* lattice $\mathcal{SIP}$ tracks a boolean valued predicate for pairs of consecutive characters. That is, the predicate is of the form $\rho : \texttt{Char} \times \texttt{Char} \to \texttt{Bool}$, where the two characters are adjacent inside the string. We can instantiate the lattice with predicates like the following:

- Gemination - whether two consecutive characters are the same. E.g. in the property names `__defineGetter__` and `__defineSetter__` there are three geminations, one for the preceding underscores, one for the double t's and one for the succeeding underscores.
- Inversions - whether two consecutive characters are inverted with respect to their lexicographical ordering. E.g. in the property name `valueOf` the characters `v` and `a` are inverted. If no characters may be inverted then the characters in the string must be sorted.

The sliding index predicate lattice is similar to the index predicate lattice. However, in addition to may- and must- bitsets and the length interval lattice, it must be equipped with the prefix-suffix lattice. This lattice is required for the concatenation operation: When s_1 and s_2 are concatenated the prefix-suffix is used to evaluate the predicate for the last character of s_1 and the first character of s_2 thus ensuring that knowledge of the predicate is preserved for all consecutive pairs of characters in the resulting string.

3.9 Prefix Suffix Inclusion ⋆

The *prefix-suffix inclusion* lattice $\mathcal{PSI}$ is inspired by the prefix-suffix and character inclusion lattices. It tracks the *set of characters* that the first and last character in the string *may* or *must* be. As for the character inclusion lattice, it tracks whether the string is the empty string or if the prefix/suffix may be a non-representable character symbol. Its representation is based on no less than *four* bitsets: May- and must- bitsets for both the prefix and suffix character. Equal, concatenation and join is implemented as bitwise operations in $\mathcal{O}(1)$ time.

The prefix-suffix inclusion lattice can rule out equality of the concrete string `prototype` and the abstract string $\hat{s}$, if the first character of $\hat{s}$ is *definitely not* `p` or the last character of $\hat{s}$ is *definitely not* `e`.

3.10 String Hash ⋆

The *string hash* lattice $\mathcal{SH}$ lattice is inspired by the length hash lattice, but instead of hashing the string length, it hashes the string itself: It uses a hash function $h : S \to U$ which takes the sum of all character codes in the string and hashes it into a bucket (as described in Section 3.4). The strength of the string hash lattice is that it can keep separate strings for which the other lattices might lose all information. Consider the example:

$$\texttt{"foo"} \hat{=} (\texttt{"The"} \sqcup \texttt{"quick"} \sqcup \texttt{"brown"} \sqcup \texttt{"fox"})$$

Here, for instance, the length interval, the length hash and the character inclusion lattices lose information and cannot rule out that the strings may be equal. In contrast, the four strings hash to 33, 29, 40 and 13, respectively, and `"foo"` hash to 4, and thus the abstraction is able to rule out equality between the left and right side. We implement the string hash lattice as a single bitset which supports equal, concatenate and join in $\mathcal{O}(1)$ time. Concatenation is implemented in the same way as the length hash lattice and requires the hash function to be distributive (see Section 3.4).

3.11 Number Strings ⋆

In JavaScript it is common for numbers to be coerced to strings. We introduce the number string lattice $\mathcal{N}$ to track JavaScript numbers encoded as strings. It is the powerset lattice of the elements ∞, $-\infty$, `NaN`, $\mathbb{N}$ and `Other` ordered by subset-inclusion:

$$\mathcal{N} = (\mathcal{P}\{\infty, -\infty, \texttt{NaN}, \mathbb{N}, \texttt{Other}\}, \subseteq)$$

Here ∞ represents the number "positive infinity" which coerced to a string yields `"Infinity"`, similarly $-\infty$ coerces to `"-Infinity"`, `NaN` represents "not-a-number" which coerces to `"NaN"` and $\mathbb{N}$ which represents any natural number which coerces to itself as a string. The number string lattice is implemented as a bitset and supports equal, concatenate and join operations in $\mathcal{O}(1)$ time. With respect to concatenate, we take a pragmatic approach and let it return $\top$, i.e. all lattice elements.

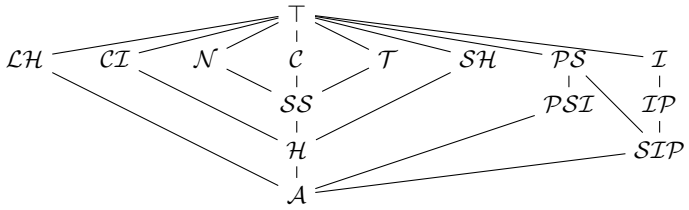

Fig. 4: A diagram showing how the precision of the lattices relate to each other. As an example, the precision of the prefix-suffix lattice $\mathcal{PS}$ lattice is fully subsumed by the prefix-suffix inclusion lattice $\mathcal{PSI}$.

3.12 Type Strings ⋆

In JavaScript the `typeof` operator inspects the runtime type of a value and returns one of the string constants: `boolean`, `function`, `object`, `string` and `undefined`. The `typeof` operator is widely used in jQuery, for instance[4]:

```
stop: function(type, clearQueue, gotoEnd) { // ...
    if (typeof type !== "string") { // ...
```

Here the behaviour of the `stop` function depends on the type of its first argument. We introduce the type string lattice $\mathcal{T}$ to explicitly track the five strings returned by `typeof`:

$$\mathcal{T} = (\mathcal{P}(\{\texttt{Bool}, \texttt{Func}, \texttt{Obj}, \texttt{Str}, \texttt{Undef}, \texttt{Other}\}), \subseteq)$$

The `Other` element, as for the number string lattice, represents all strings other than the type strings. We implement the lattice as a single bitset which supports the equal and join operations in $\mathcal{O}(1)$ time.

3.13 The Hybrid Lattice ⋆

We introduce the hybrid string lattice $\mathcal{H}$ as the cartesian product of the string set $\mathcal{SS}$ (Section 3.2), character inclusion $\mathcal{CI}$ (Section 3.6) and string hash $\mathcal{SH}$ (Section 3.10) lattices. The intuitive idea behind the lattice is to track a few concrete strings with full precision and then "fallback" to the character inclusion and string hash lattices when there are too many strings to track. As will be shown in Section 4, the hybrid lattice achieves almost the same precision as the combination of all presented lattices.

3.14 Lattice Relations

Figure 4 shows how the precision of the lattices relate to each other. As discussed in the previous section, the figure shows that the hybrid string lattice $\mathcal{H}$ is at least as precise as the string set $\mathcal{SS}$, character inclusion $\mathcal{CI}$ and string hash lattices $\mathcal{SH}$. We call the cartesian product of all lattices $\mathcal{A}$.

[4] jQuery v1.8.3, line 9,046

	$\mathcal{C}$	$\mathcal{SS}$	$\mathcal{I}$	$\mathcal{LH}$	$\mathcal{PS}$	$\mathcal{CI}$	$\mathcal{IP}$	$\mathcal{PSI}$	$\mathcal{SH}$	$\mathcal{N}$	$\mathcal{T}$	$\mathcal{H}$
New				✓			✓	✓	✓	✓	✓	✓
Structural	✓	✓			✓		✓	✓		✓	✓	✓
Subset		✓		✓		✓	✓	✓	✓	✓	✓	✓
Parametric						✓	✓	✓	✓			✓
Space	$\lvert s\rvert$	$k \times \lvert s\rvert$	2	1	2	2	4	5	1	1	1	$k \times \lvert s\rvert + 3$

Table 1: Overview of lattice characteristics.

3.15 Overview

We briefly summarize some characteristics of the presented lattices:

New We believe that the lattice is new to the literature.
Structural The lattice tracks the structure of the string. As an example, the prefix-suffix lattice $\mathcal{PS}$ tracks the first and last character of the string.
Subset The lattice subset or superset-based.
Parametric The lattice has different instantiations. For instance, the index predicate lattice $\mathcal{IP}$ can be instantiated with different predicates.
Space The space (memory) required to represent a single lattice element. In machine words, except for $\mathcal{C}$ and $\mathcal{SS}$ which must store the entire string(s).

Table 1 shows an overview of these characteristics.

3.16 Additional String Operations

We now describe some additional string operations which the lattices support.

charAt and charCodeAt The *charAt*(i) and *charCodeAt*(i) string functions return the character or character code at position i inside the string.

- $\mathcal{PS}$– if the index is zero then the prefix-suffix lattice knows the precise result.
- $\mathcal{IP}$– the index predicate lattice can provide an upper bound on what character symbols may occur at index i. E.g. if the predicate is `isUpperCase` and it holds for index i, then the character must be in the set $[A - Z]$.
- $\mathcal{CI}$– the character inclusion lattice can provide an upper bound on what character symbols may occur at index i.

indexOf, lastIndexOf and search The *indexOf*(s) and *lastIndexOf*(s) functions return the index of respectively the first and last occurrence of s in the string. If s is not contained in the string, the value -1 is returned.

- $\mathcal{CI}$– if the query string is a single character the character inclusion lattice can decide whether that character may or must occur within the string. It cannot give the precise index, but it can decide whether the -1 value should be part of the return value.

– $\mathcal{IP}$– if the query string is a single character and some property of that character is tracked by the index predicate lattice, then a set of indices can be returned. E.g. if the query string is an uppercase `A` and the index predicate lattice tracks uppercase letters, then the lattice can provide all indices where uppercase letters may occur.
– $\mathcal{I}$– the length interval lattice can provide a bound on the returned index.

substring The *substring(b, e)* function returns the substring beginning at position b and ending immediately before position e.

– $\mathcal{I}$ & $\mathcal{LH}$– the length interval and length hash lattices simply restrict their intervals to the range $[b, e]$.
– $\mathcal{PS}$– if the extracted string is a prefix, i.e. if $b = 0$, then the prefix-suffix lattice can retain its first component.
– $\mathcal{CI}$– the character inclusion lattice can retain its *may-set* of character symbols, but its *must-set* must be replaced by $\top$.
– $\mathcal{IP}$ & $\mathcal{SIP}$– the index predicate and sliding index predicate lattices can retain all their information for the substring.

4 Evaluation

We have described the theoretical properties of the lattices and now turn to their practical application by considering the research questions:

– **Q1**: How precise are the lattices, independent of any particular analysis, for reasoning about strings used in dynamic property accesses?
– **Q2**: To what degree does a more precise string lattice, for dynamic property accesses, improve the overall precision and performance of a static analysis?

4.1 Dynamic Analysis

We investigate Q1 by performing a dynamic analysis of strings and dynamic property accesses in four large JavaScript libraries. Inspired by Liang et al. [12] the dynamic analysis is used to provide a (static-) analysis independent upper bound on the precision of each lattice. That is, the best precision each lattice can possibly provide for a set of concrete execution traces.

We instantiate the string set lattice with $k = 3$ (see Section 3.2), the length interval lattice with width $k = 20$ (see Section 3.3), the index predicate lattice with the uppercase predicate (see Section 3.7) and the remaining lattices are instantiated as described in their respective sections.

Benchmarks We collect concrete execution traces for the four large JavaScript libraries shown in Table 5. The traces expose a total of 80,000 dynamic property accesses of which 60,000 are reads. We obtained the traces by loading twelve

		Reads		Writes	
Library	Lines	Locations	Properties	Locations	Properties
jQuery-1.9.1	9,597	400	7.0	124	5.8
jQuery-1.8.1	9,301	377	12.0	102	8.3
jQuery-1.7.1	9,266	401	6.8	101	6.6
Prototype-1.7.0	7,036	226	9.8	43	14.7
MooTools-1.4.5	5,976	281	13.7	110	14.2
jQueryUI-1.8.24	11,377	265	8.1	75	7.4

Fig. 5: The JavaScript libraries used for the dynamic analysis evaluation. Here the *locations* column indicates the number of syntactic occurrences of dynamic property accesses. The *properties* column indicates the average number of property names read-/written by a dynamic property access expression.

popular websites according to the Alexa rankings[5]. The complete list of websites is available in Appendix A. Since jQuery is prevalent, we include three different versions. The websites are automatically modified to use instrumented versions of the libraries which record information about every dynamic property access.

We explain Table 5 by example. The table shows that the jQuery-1.9.1 source code has 400 dynamic property read expressions and 124 write expressions. For the read expressions, an average of 7.0 properties are read by each expression, and an average of 5.8 properties are written by each expression.

Concrete Traces We instrument the source code to register the following for every dynamic property access `o[p]`:

$$\mathcal{T} = (\mathcal{R}, \mathcal{L}, \mathcal{E}, \mathcal{P}_o, \mathcal{P}_p) \text{ , where}$$

- $\mathcal{R}$ is a unique identifier for the concrete *run*.
- $\mathcal{L}$ is the physical *location* of the dynamic property access in the source code.
- $\mathcal{E}$ is the *expression tree* corresponding to how the property name, which is being used for the dynamic property access, was created. An expression tree is a tree where the leaves are string constants and the internal nodes are string operations, which are equipped with their source code location.
- $\mathcal{P}_o$ is the set of properties available on the object o itself.
- $\mathcal{P}_p$ is the set of properties available on the prototype objects of o.

Here $\mathcal{R}$ and $\mathcal{L}$ is meta data about the concrete trace and $\mathcal{E}, \mathcal{P}_o, \mathcal{P}_p$ is information about the dynamic property access. As an example, the execution of the code snippet on the left produces the trace on the right.

Here `Toplevel` is the name of the toplevel "function", $\mathcal{E}$ is the expression tree for the string concatenation of `"p"` and `"q"`, $\mathcal{P}_o$ contains `a` and `b` (i.e all properties of `o`) and $\mathcal{P}_p$ contains all properties of the `Object[[prototype]]` object.

[5] http://www.alexa.com/topsites

```
x = new Object();
x.a = 42;
x.b = 21;
z = x["p" + "q"];
```

$$\mathcal{T} = (0, \texttt{input.js:4}, \texttt{Toplevel}, \mathcal{E}, \mathcal{P}_o, \mathcal{P}_p)$$
$$\mathcal{E} = \texttt{Concat(input.js:4, "p", "q")}$$
$$\mathcal{P}_o = \{\texttt{a}, \texttt{b}\}$$
$$\mathcal{P}_p = \{..., \texttt{toString}, \texttt{valueOf}, ...\}$$

Abstract Traces We simulate the effects of abstraction by merging several concrete traces into a smaller set of abstract traces. We merge concrete traces which share the same location $\mathcal{L}$ to obtain a single abstract trace for that location. In particular, given two concrete traces $\mathcal{T}^1 = (\mathcal{R}^1, \mathcal{L}^1, \mathcal{E}^1, \mathcal{P}_o^1, \mathcal{P}_p^1)$ and $\mathcal{T}^2 = (\mathcal{R}^2, \mathcal{L}^2, \mathcal{E}^2, \mathcal{P}_o^2, \mathcal{P}_p^2)$ we define the abstract trace $\hat{\mathcal{T}} = (\mathcal{L}, \hat{\mathcal{E}}, \hat{\mathcal{P}}_o, \hat{\mathcal{P}}_p)$ where

$$\mathcal{L} = \mathcal{L}^1 = \mathcal{L}^2 \qquad \hat{\mathcal{E}} = \{\mathcal{T}_{\mathcal{E}}^1, \mathcal{T}_{\mathcal{E}}^2\} \qquad \hat{\mathcal{P}}_o = \mathcal{T}_{\mathcal{P}_o}^1 \cup \mathcal{T}_{\mathcal{P}_o}^2 \qquad \hat{\mathcal{P}}_p = \mathcal{T}_{\mathcal{P}_p}^1 \cup \mathcal{T}_{\mathcal{P}_p}^2$$

The generalization to multiple traces is straightforward.

We now consider two scenarios. First, we evaluate the precision of the lattices on the abstract traces where the expression tree $\mathcal{E}$ is fully evaluated before any abstraction. That is, if an abstract trace has the two expressions trees $\mathcal{E}_1 = "a"$ and $\mathcal{E}_2 = "b" + "c" + "d"$ then we consider the abstraction $\alpha("a") \sqcup \alpha("bcd")$, i.e. concatenation occurs *before* abstraction. Second, we evaluate the precision when concatenation occurs *after* abstraction. For instance, we would evaluate $\alpha("a") \sqcup (\alpha("b") + \alpha("c") + \alpha("d"))$. Here α is the abstraction function which lifts a concrete string into the abstract domain.

Precision without Concatenation Figure 6 shows the percentage of dynamic property access locations with *zero* false positives for each lattice. That is, a value of 100% implies that the lattice is complete for all dynamic property accesses. A value of 50% implies that half of all locations of dynamic property accesses have at least one false positive. The figure shows two bars for each lattice; the light bar represents locations with false positives involving properties in the base object, and the dark bar represents false positives involving properties in the prototype objects.

We observe that the constant string lattice ensures that at most 50% of all dynamic property accesses involving base object properties have zero false positives. If we consider prototype properties, the number drops to 31%. This means that for more than half of all dynamic property accesses the constant string lattice will cause spurious flow. For the string set lattice the percentages are not surprisingly higher at 72% and 58%, respectively. The character inclusion lattice achieves the highest precision with 79% and 78% of all dynamic property accesses having zero false positives. Remarkably, the prefix-suffix inclusion lattice achieves nearly the same precision, even though it only tracks information about the first and last characters in the string. The hybrid lattice achieves 89% and 86% which is only slightly lower than the all lattice (the product of all lattices). The number string and type string lattices achieve less than 25% of property accesses with zero false positives and are omitted from the graphs.

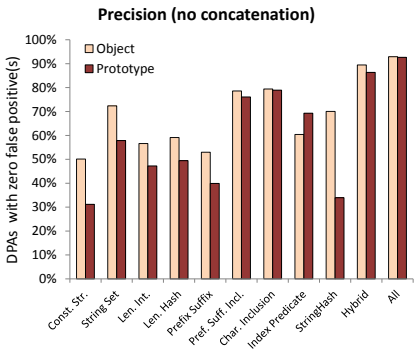

Fig. 6: Precision *without* concatenation, measured as the number of dynamic property accesses with zero false positives.

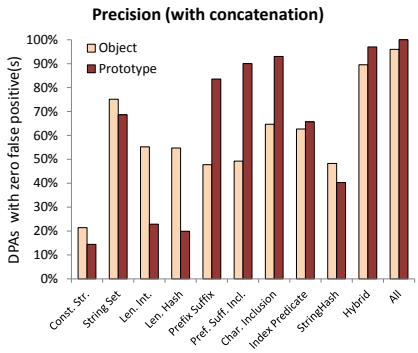

Fig. 7: Precision *with* concatenation, measured as the number of dynamic property accesses with zero false positives.

We attribute the difference in precision for object and prototype properties to the fact that most objects have fewer properties than their corresponding prototype object(s).

Precision with Concatenation Figure 7 shows the percentage of dynamic property accesses with *zero* false positives for each lattice restricted to traces which involve at least one concatenation. This restriction reduces the number of concrete traces from 80,000 to around 8,000. Note that this implies that Figures 6 and 7 are not directly comparable.

We observe, when concatenation is involved, the constant string lattice is only able to achieve a zero false positive rate of 21% and 14% for object and prototype properties. Again, the character inclusion lattice achieves the best precision with 64% and 93% property accesses with zero false positives. The hybrid lattice achieves 90% and 97% which is only slightly less than all the lattices combined. The number string and type string lattices achieve less than 10% with zero false positives and are omitted from the graphs.

We leave the evaluation of the sliding index predicate as future work for two reasons: First, initial experiments on the index predicate lattice showed that it has very poor performance for concatenation. Second, it was not clear to us what kind of predicate would be a good discriminator for property names.

We answer Q1 by concluding that the precision of the constant string lattice is worse than most other of the presented lattices. Furthermore, the hybrid string lattice $\mathcal{H}$ achieves almost the same precision as all the lattices combined.

4.2 Static Analysis

We investigate Q2 by comparing the precision and performance of a static analysis equipped with the constant string lattice $\mathcal{C}$ and the proposed hybrid string lattice $\mathcal{H}$.

			Precision		Performance		
Program	Lines	Nodes	PAs↓	PointsTo↓	Constant	Hybrid	Speedup
3d-cube.js	343	2,794	14%	10%	1.7s	1.0s	1.6x
3d-raytrace.js	443	2,874	57%	41%	38.5s	4.7s	8.2x
access-nbody.js	170	828	9%	7%	0.3s	0.2s	2.1x
astar.js	355	1,406	68%	58%	5.9s	0.3s	16.8x
crypto-md5.js	295	1,422	93%	93%	0.3s	0.2s	1.8x
garbochess.js	2,812	15,795	78%	77%	56.3s	24.3s	2.3x
javap.js	1,400	5,104	28%	27%	7.5s	7.3s	1.0x
richards.js	541	1,602	3%	2%	3.7s	3.3s	1.1x
simplex.js	450	2,056	73%	72%	0.6s	0.3s	2.0x
splay.js	398	1,016	2%	2%	0.4s	0.4s	1.0x

Table 2: Static Analysis results. Lines is the number of lines of source code. Nodes is the number of control-flow graph nodes. PAs↓ is the percentage of property reads with improved precision. PointsTo↓ is the average reduction in the size of points-to sets for *all* property reads.

Dataflow Analysis We have implemented an inter-procedural, flow-sensitive and context-insensitive dataflow analysis for JavaScript in the style of Jensen et al. [10]. The analysis can be instantiated with different string lattices without any changes to the rest of the abstraction.

Benchmarks We evaluate the analysis on the programs shown in Table 2. The `3d-cube.js`, `3d-raytrace.js`, `access-nbody.js` and `crypto-md5` programs originate from the Mozilla SunSpider benchmark suite, `richards.js` and `splay.js` originate from the Google Octane benchmark suite and `astar.js`, `garbochess.js`, `javap.js` and `simplex.js` were collected from GitHub and various sources on the Internet. The table lists the benchmark name, number of lines of code and the number of control-flow graph nodes in the first three columns. We use these benchmarks, instead of the libraries from the previous section, since we know of no analysis which is yet able to analyze such large and complex libraries.

Precision We compare the precision of the string lattices in two ways. First, we compute for how many property read locations that the points-to sets are smaller. Second, we compute on average how much smaller the points-to sets are. We look at all property reads and not just dynamic property reads. The reason is that spurious flow in one dynamic property access may cause imprecision in a non-dynamic read. Thus, by looking at all reads we get a clearer picture of overall analysis precision.

The PAs↓ column in Table 2 shows the percentage of property reads where the use of the hybrid string lattice results in a smaller points-to set than the constant string lattice, that is, the percentage of reads where the hybrid string lattice yields at least one less pointer than the constant string lattice. The results show that for 5 of the 10 programs at least 50% of all property reads have improved precision, and that all programs show some improvement. The PointsTo↓ column

shows how much smaller on average the points-to sets are for all property reads. The results show that the hybrid lattice ensures significantly smaller sets and that for 5 of the 10 programs the reduction is more than 40%. Thus the hybrid lattice improves precision for many property accesses and is effective at reducing spurious flow compared to the constant string lattice.

Performance The last three columns of Table 2 compare the analysis time with the two different lattices. The results show that for 7 of the 10 programs the analysis is more than 1.5x faster, and for 5 of the programs the analysis is more than 2.0x faster. We attribute this to the fact that the analysis is more precise with the hybrid lattice and propagates less spurious flow. In the case of `javap.js`, `richards.js` and `splay.js` there is no significant speedup. In case of `richards.js` and `splay.js` this can be explained by the fact that these two benchmarks gain little in terms of improved precision. The `javap.js` program appears to be an outlier which gains significantly improved precision, but no corresponding boost in performance. Naturally, the degree of speedup will vary from analysis to analysis. In particular, if the analysis is efficient at representing and propagating large point-to sets the performance improvement will likely be less pronounced.

We answer Q2 by concluding that the hybrid string lattice $\mathcal{H}$ is preferable to the commonly used constant string lattice $\mathcal{C}$. We have shown that the hybrid string lattice leads to significantly improved precision and performance.

5 Conclusion

We have described twelve different string abstractions – five previously known and seven new – for reasoning about dynamic property accesses in static analysis of JavaScript. Experimental evaluation on four common and large JavaScript libraries, including jQuery, suggests that dynamic property accesses are prevalent and that the standard approach of tracking strings with the constant string lattice is insufficient. We have presented the hybrid lattice $\mathcal{H}$ which supports the equal, concatenate and join operations in $\mathcal{O}(1)$ time. Experimental results on 10 JavaScript programs show that the hybrid string lattice leads to significantly improved precision and performance when used in a dataflow analysis.

References

1. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
2. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *SAS*, pages 1–18, 2003.
3. Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged Information Flow for JavaScript. In *PLDI*, pages 50–62, 2009.
4. Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. Static Analysis of String Values. In *ICFEM*, pages 505–521, 2011.
5. Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.
6. Salvatore Guarnieri and V. Benjamin Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, pages 151–168, 2009.
7. Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the World Wide Web from Vulnerable JavaScript. In *ISSTA*, pages 177–187, 2011.
8. Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the Eval that Men Do. In *ISSTA*, pages 34–44, 2012.
9. Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, September 2011.
10. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium, SAS*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.
11. Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A Solver for String Constraints. In *ISSTA*, pages 105–116, 2009.
12. Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. A Dynamic Evaluation of the Precision of Static Heap Abstractions. In *OOPSLA*, pages 411–427, 2010.
13. Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *APLAS*, pages 307–325, 2008.
14. Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *PLDI*, pages 1–12, 2010.
15. Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP*, pages 435–458, 2012.
16. Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. In *ESEC/SIGSOFT FSE*, pages 114–124, 2013.

A Appendix

The concrete traces were scraped from the following websites:

URL	Library	Version
http://jquery.com/	jQuery	1.9.1
http://www.chacha.com/	jQuery	1.9.1
http://themeforest.net/	jQuery	1.8.1
http://www.guardian.co.uk/	jQuery	1.8.1
http://adf.ly/	jQuery	1.7.1
http://stackoverflow.com/	jQuery	1.7.1
http://www.fixya.com/	jQuery UI	1.8.24
http://www.goal.com/en-us/	jQuery UI	1.8.24
http://www.6.cn/	MooTools	1.4.5
http://www.aeriagames.com/	MooTools	1.4.5
http://hubpages.com/	Prototype	1.7.0
http://www.last.fm/	Prototype	1.7.0