Detecting Javascript Vulnerabilities Through the Abstract Interpretation of Strings

Christian Maldonado
Koby Picker
Dr. Lunjin Lu

Outline

- Background
  - Problem Definition
  - Related Work
  - Dataflow Analysis, Abstract Interpretation, Automata
- TAJS
- Widening

Background

Abstract. JavaScript is ubiquitous online. It's important then, that web applications which use JavaScript are secure against common string manipulation exploits such as SQL injections and cross-site scripting attacks. JavaScript is dynamically typed, and objects can have their methods and properties dynamically modified at runtime -- thus, static analysis tools are a challenge to implement and security analysis tools are consequently limited. We propose the abstract interpretation of JavaScript strings into finite state automata. We augment TAJS (Type Analyzer for JavaScript), an open source dataflow analysis tool for JavaScript, in order to precisely approximate strings in real-world JavaScript code, and warn against common vulnerabilities.

The Problem

Javascript is hard to analyze, which often leaves web-applications vulnerable to exploits.

Related Work

Yu, Fang, Muath Alkhalaf, and Tevfik Bultan. "Stranger: An automata-based string analysis tool for PHP." In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 154-157. Springer Berlin Heidelberg, 2010.

Costantini, Giulia, Pietro Ferrara, and Agostino Cortesi. "Static analysis of string values." In *International Conference on Formal Engineering Methods*, pp. 505-521. Springer Berlin Heidelberg, 2011.

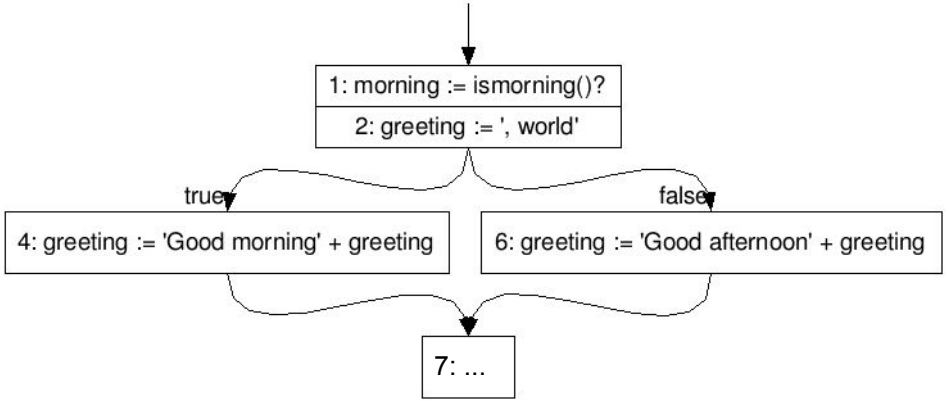
Yu, Fang, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. "Symbolic string verification: An automata-based approach." In *International SPIN Workshop on Model Checking of Software*, pp. 306-324. Springer Berlin Heidelberg, 2008.

Minamide, Yasuhiko. "Static approximation of dynamically generated web pages." In *Proceedings of the 14th international conference on World Wide Web*, pp. 432-441. ACM, 2005.

Jensen, Simon Holm, Anders Møller, and Peter Thiemann. "Type analysis for JavaScript." In *International Static Analysis Symposium*, pp. 238-255. Springer Berlin Heidelberg, 2009.

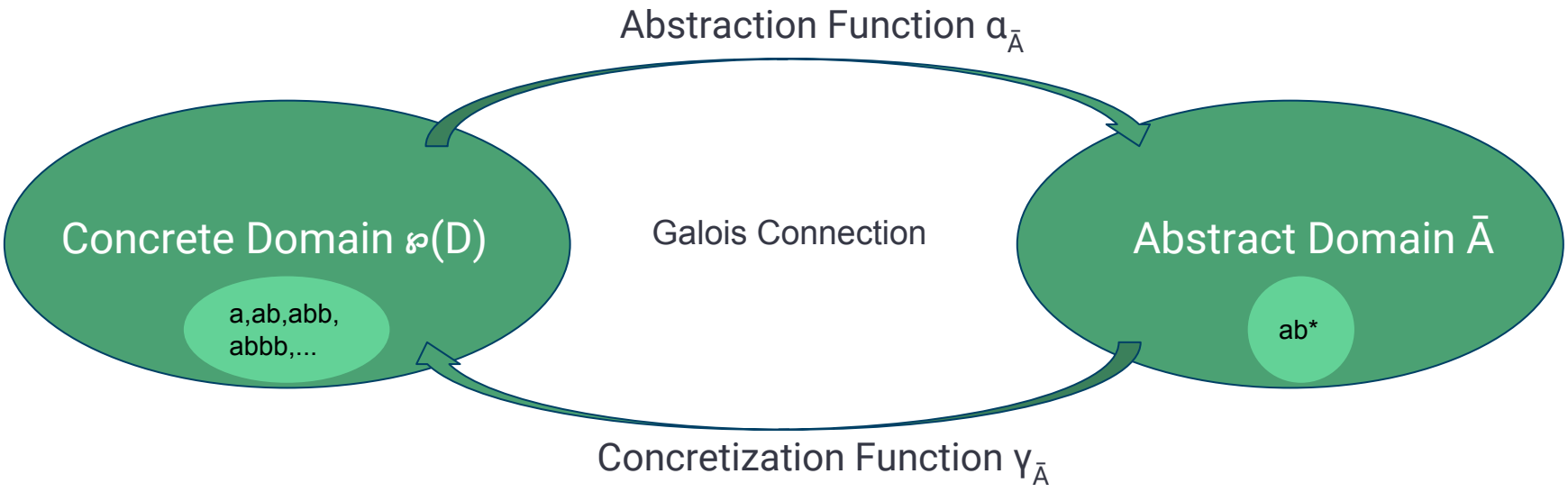
Dataflow Analysis, Abstract Interpretation, Automata

```
1: morning ← ismorning?()
2: greeting ← ", world!"
3: if morning then
4:     greeting ← "Good morning" + greeting.
5: else
6:     greeting ← "Good afternoon" + greeting.
```



What is the value of *greeting* after the ***if-else***?

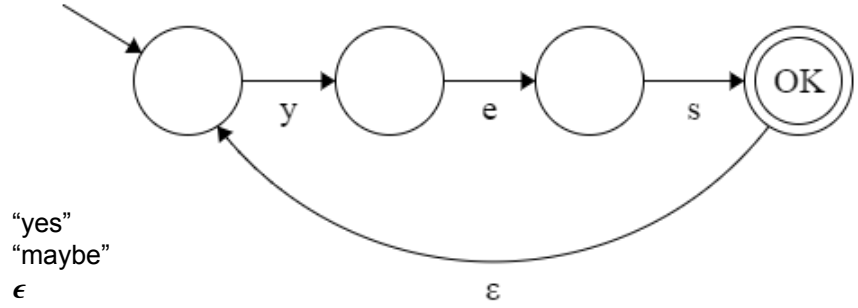
Dataflow Analysis, **Abstract Interpretation**, Automata



Dataflow Analysis, Abstract Interpretation, **Automata**

- Automata are machines that can interpret *regular languages*
- They are equivalent to *regular expressions*
- Also called DFAs (Deterministic Finite Automata) or NFAs (Nondeterministic).
- We might use these terms interchangeably (even though we really shouldn't!)

Definition 1 *An Automata A is given by:*
$A = \{Q, q_0, \Sigma, \delta, F\}$ *where* Q *is a set of states,* q_0 *is the start state,* Σ *is your alphabet,* δ *is a list of transitions, and* F *is a list of final or accepting states*



"yes"
"maybe"
ϵ

Type Analysis for Javascript

Abstract. We present a static program analysis infrastructure that can infer detailed and sound type information for JavaScript programs using abstract interpretation. The analysis results can be used to detect common programming errors – or rather, prove their absence, and for producing type information for program comprehension. Preliminary experiments conducted on real-life JavaScript code indicate that the approach is promising regarding analysis precision on small and medium size programs, which constitute the majority of JavaScript applications. With potential for further improvement, we propose the analysis as a foundation for building tools that can aid JavaScript programmers. (Jensen et. al.)

An example

```
1   var z = "h" + "i";
2   var cond = Math.random() > .5;
3
4   function greeting () {
5       var x;
6       if (cond) {
7           x = "hello";
8       } else {
9           x = "goodbye";
10      }
11      return x;
12  }
13  var w = greeting();
14  var y = w + "something, world!".substring(9);
```

TAJS - Before DFA Implementation

```
1   var z = "h" + "i";
2   var cond = Math.random() > .5;
3
4   function greeting () {
5       var x;
6       if (cond) {
7           x = "hello";
8       } else {
9           x = "goodbye";
10      }
11      return x;
12  }
13  var w = greeting();
14  var y = w + "something, world!".substring(9);
```

```
z:  /home/c                                                    1.js:1:9   ->  "hi"
Math:   /ho                                                    ris_1.js:2:12  ->  [@Math[native]]
random: /ho                                                    ris_1.js:2:12  ->  [@Math.random[native]]
cond:   /ho                                                    ris_1.js:2:12  ->  Bool
greeting:                                                      s/chris_1.js:13:9  ->  [@greeting#fun1]
w:  /home/c                                                    1.js:13:9  ->  IdentStr
w:  /home/chris/Documents/Enero2016/REU/TAJS-master/test/chris/chris_1.js:14:9  ->  IdentStr
substring:  /home/chris/Documents/Enero2016/REU/TAJS-master/test/chris/chris_1.js:14:13 ->  [@String.prototype.substring[native]]
y:  /home/chris/Documents/Enero2016/REU/TAJS-master/test/chris/chris_1.js:14:9  ->  Str
cond:   /home/chris/Documents/Enero2016/REU/TAJS-master/test/chris/chris_1.js:6:9   ->  Bool
x:  /home/chris/Documents/Enero2016/REU/TAJS-master/test/chris/chris_1.js:7:9   ->  "hello"
x:  /home/chris/Documents/Enero2016/REU/TAJS-master/test/chris/chris_1.js:9:9   ->  "goodbye"
x:  /home/chris/Documents/Enero2016/REU/TAJS-master/test/chris/chris_1.js:11:12 ->  IdentStr

Process finished with exit code 0
```

TAJS - After DFA Implementation

```
1   var z = "h" + "i";
2   var cond = Math.random() > .5;
3
4   function greeting () {
5       var x;
6       if (cond) {
7           x = "hello";
8       } else {
9           x = "goodbye";
10      }
11      return x;
12  }
13  var w = greeting();
14  var y = w + "something, world!".substring(9);
```

```
z:  /home/                                                    ->  'hi'
Math:   /h                                                    s:2:12 ->  [@Math[native]]
random: /h                                                    s:2:12 ->  [@Math.random[native]]
cond:   /h                                                    s:2:12 ->  Bool
greeting:                                                     st.js:13:9 ->  [@greeting#fun1]
w:  /home/                                                    :9 ->  ['goodbye', 'hello']
w:  /home/chris/Documents/Enero2016/REU/TAJS/test/koby/test.js:14:9 ->  ['goodbye', 'hello']
substring:  /home/chris/Documents/Enero2016/REU/TAJS/test/koby/test.js:14:13    ->  [@String.prototype.substring[native]]
y:  /home/chris/Documents/Enero2016/REU/TAJS/test/koby/test.js:14:9 ->  ['goodbye, world!', 'hello, world!']
cond:   /home/chris/Documents/Enero2016/REU/TAJS/test/koby/test.js:6:9  ->  Bool
x:  /home/chris/Documents/Enero2016/REU/TAJS/test/koby/test.js:7:9  ->  'hello'
x:  /home/chris/Documents/Enero2016/REU/TAJS/test/koby/test.js:9:9  ->  'goodbye'
x:  /home/chris/Documents/Enero2016/REU/TAJS/test/koby/test.js:11:12    ->  ['goodbye', 'hello']

Process finished with exit code 0
```

Widening

Since DFAs can represent infinite sets of strings, the fixpoint computations are not guaranteed to converge. To alleviate this problem, we use the automata widening technique proposed by Bartzis and Bultan [4] to compute an over- approximation of the least fixpoint. Briefly, we merge those states belonging to the same equivalence class identified by certain conditions. (Fang, et. al.)

Bartzis and Bultan's Widening Operator

Given two finite automata $A = (K, \Sigma, \delta, e, F)$ and $A' = (K', \Sigma, \delta', e', F')$ we define the binary relation $\equiv_{\nabla}$ on $K \cup K'$ as follows. Given $k \in K$ and $k' \in K'$, we say that $k \equiv_{\nabla} k'$ and $k' \equiv_{\nabla} k$ if and only if

$$(1) \quad \forall w \in \Sigma^*. \delta^*(k, w) \in F \Leftrightarrow \delta'^*(k', w) \in F'$$

$k \equiv_{\nabla} k'$ if the languages accepted by A from k and by A' from k' are the same…. $L(A_k) = L(A'_{k'})$

$$(2) \quad \text{or } k, k' = \text{sink} \wedge \exists w \in \Sigma^*. \delta^*(e, w) = k \wedge \delta'^*(e', w) = k',$$

$k \equiv_{\nabla} k'$ if for some word w, A ends up in state k and A' ends up in state k' after consuming w.

where $\delta^*(k, w)$ is defined as the state A reaches after consuming w starting from state k.

Bartzis and Bultan's Widening Operator

For $k_1 \in K$ and $k_2 \in K$ we say that $k_1 \equiv_{\nabla} k_2$ if and only if

$$\exists k' \in K'.\ k_1 \equiv_{\nabla} k' \wedge k_2 \equiv_{\nabla} k' \quad \vee \quad \exists k \in K.\ k_1 \equiv_{\nabla} k \wedge k_2 \equiv_{\nabla} k$$

k_1 and k_2 are equivalent to the same state in K'

k_1 and k_2 are equivalent to the same state in K

Similarly we can define $k'_1 \equiv_{\nabla} k'_2$ for $k'_1 \in K'$ and $k'_2 \in K'$.

Bartzis and Bultan's Widening Operator

Call C the set of equivalence classes of $\equiv_{\nabla}$. We define $A \nabla A' = (K'', \Sigma, \delta'', e'', F'')$ by:

$K'' = C$

The set of states of $A \nabla A'$ is the set C of equivalence classes of $\equiv_{\nabla}$.

$\delta''(c_i, \sigma) = c_j$ s.t. $(\forall k \in c_i \cap K.\ \delta(k, \sigma) \in c_j \vee \delta(k, \sigma) = sink) \wedge (\forall k' \in c_i \cap K'.\ \delta(k', \sigma) \in c_j \vee \delta(k', \sigma) = sink)$

Transitions are defined from the transitions of A and A'.

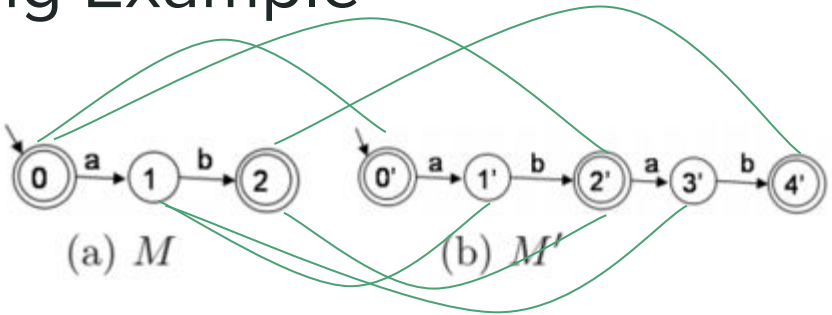
$e'' = c$ s.t. $e \in c \wedge e'' \in c$

The initial state is the class containing the initial states e and e'.

$F'' = \{c_1, c_2, ..., c_n\}$ s.t. $\forall c_i,\ \exists k \in F \cup F'.\ k \in c_i$

The set of final states is the set of classes that contain some of the final states in F and F'.

Widening Example



L(M) = {∈, ab} L(M') = {∈, ab, abab} L(M ∇ M') = $(ab)^*$

$C = \{k''_0, k''_1\}$

$k''_0 = \{k_0, k'_0, k_2, k'_2, k'_4\}$

$k''_1 = \{k_1, k'_1, k'_3\}$

Timeline

Week 6: Continue implementations and changes

Week 7: Testing & debugging

Week 8: Start writing

Week 9: Continue writing

Week 10: Prepare for final poster presentation

References

[1] Costantini, Giulia, Pietro Ferrara, and Agostino Cortesi. "Static analysis of string values." In *Formal Methods and Software Engineering*, pp. 505-521. Springer Berlin Heidelberg, 2011.

[2] Jensen, Simon Holm, Anders Møller, and Peter Thiemann. "Type analysis for JavaScript." In Static Analysis, pp. 238-255. Springer Berlin Heidelberg, 2009.

[3] Yu, Fang, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. "Symbolic string verification: An automata-based approach." In *International SPIN Workshop on Model Checking of Software*, pp. 306-324. Springer Berlin Heidelberg, 2008.

[4] Bartzis, Constantinos, and Tevfik Bultan. "Widening arithmetic automata." In*International Conference on Computer Aided Verification*, pp. 321-333. Springer Berlin Heidelberg, 2004.