Information Flow Analysis for JavaScript via a Dynamically Typed Language with Staged Metaprogramming

Martin Lester[1] Luke Ong[1] Max Schäefer[2]

[1]Department of Computer Science, University of Oxford

[2]IBM T.J. Watson Research Center

Student Conference, 2012–11–16

Motivation

Web applications written in JavaScript regularly handle sensitive data.

- Reasoning about their security properties is an important problem.
- JavaScript is a difficult language to reason about.

Why is JavaScript difficult?

- Poorly understood, quaint semantics.
- Many features: mutable state, exceptions, dynamic types, prototype-based inheritance, type coercion, first-class functions ...

Motivation

Web applications written in JavaScript regularly handle sensitive data.

- Reasoning about their security properties is an important problem.
- JavaScript is a difficult language to reason about.

Why is JavaScript difficult?

- Poorly understood, quaint semantics.
- Many features: mutable state, exceptions, dynamic types, prototype-based inheritance, type coercion, first-class functions ... and **eval**, which is widely used.

What have we done about it?

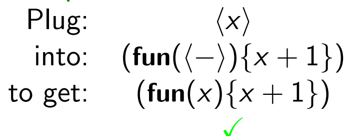
- Produced an information flow analysis for a language with many of JavaScript's features, including run-time code generation.

Staged Metaprogramming

JavaScript's **eval** is a form of metaprogramming: it allows construction, manipulation and evaluation of program code at run-time. But metaprogramming is not new:

- Lisp allows quoting and unquoting of code.
- This restricts manipulation to plugging holes in abstract syntax trees.

Example



Staged Metaprogramming

JavaScript's **eval** is a form of metaprogramming: it allows construction, manipulation and evaluation of program code at run-time. But metaprogramming is not new:

- Lisp allows quoting and unquoting of code.
- This restricts manipulation to plugging holes in abstract syntax trees.

Example

Plug:	$\langle x \rangle$	Plug:	$\langle \})(\textbf{fun}(y)\{y \rangle$
into:	$(\textbf{fun}(\langle - \rangle)\{x+1\})$	into:	$(\textbf{fun}(x)\{\langle - \rangle\})$
to get:	$(\textbf{fun}(x)\{x+1\})$	to get:	$(\textbf{fun}(x)\{\})(\textbf{fun}(y)\{y\})$
	✓		✗

Staged Metaprogramming

JavaScript's **eval** is a form of metaprogramming: it allows construction, manipulation and evaluation of program code at run-time. But metaprogramming is not new:

- Lisp allows quoting and unquoting of code.
- This restricts manipulation to plugging holes in abstract syntax trees.

Example

Plug:	$\langle x \rangle$	Plug:	$\langle \})(\textbf{fun}(y)\{y \rangle$
into:	$(\textbf{fun}(\langle - \rangle)\{x+1\})$	into:	$(\textbf{fun}(x)\{\langle - \rangle\})$
to get:	$(\textbf{fun}(x)\{x+1\})$	to get:	$(\textbf{fun}(x)\{\})(\textbf{fun}(y)\{y\})$
	✓		✗

In an attempt to understand better the behaviour of **eval**, we study a language with staged metaprogramming in the style of Lisp.

- Syntactically, staged just means that quotes can be nested.

Definition of Noninterference

Consider a setting where inputs and outputs to a program are marked with security levels, such as H for high security and L for low security.

Example

In a Web application, high/low input/output channels might be:

- high input — a “password” input box
- low input — any other text box
- high output — encrypted connection to webserver
- low output — unencrypted connection to webserver

- If the high inputs of a program cannot affect the low outputs, the program satisfies noninterference.
- This means an attacker who can only view low outputs cannot gain any information about high inputs.
- Noninterference is a popular information security property, but most verification work focuses on statically typed languages.

Information Flow

Consider the program:

$$\textbf{if}(h)\{\textbf{true}\}\,\textbf{else}\{l\}$$

The result of the program can be either **true** or l.

- As the value l can flow directly to the result, we say that there is a direct flow from l to the result.
- If l is **false**, then the result of the program is equal to h. As this dependency arises only through control flow, we say that there is an indirect flow from h to the result.
- An information flow analysis tells us, for any variable x, whether it used in the computation of another variable y.
- Alternatively, in our setting with marked security levels, we can check whether any value labelled with level H is used to compute other variables (or the result of a program).
- We can use an information flow analysis to verify noninterference.

Outline

Motivation
- Metaprogramming
- Noninterference and Information Flow

Outline

Information Flow Analysis for SLamJS
- The Language SLamJS
- CFA for SLamJS
- Information Flow for SLamJS

Future Work

Conclusion

About SLamJS

Our JavaScript-like language is called SLamJS. It has:

- a dynamic type system,
- first-class functions,
- staged metaprogramming and
- objects with prototype-based inheritance.

An example:

$$\begin{array}{ll} & ((\textbf{fun}(x)\{(\textbf{fun}(y)\{x\})\})(1))(2) \\ \rightarrow & (\textbf{fun}(y)\{1\})(2) \\ \rightarrow & 1 \end{array}$$

SLamJS has explicit security level markers $\mathsf{L}, \mathsf{H}, \ldots$ that we can put on expressions:

- $\textbf{if}(\mathsf{H} : x)\{\textbf{true}\}\,\textbf{else}\{\mathsf{L} : y\}$

Staged Metaprogramming in SLamJS

SLamJS allows staged metaprogramming with these constructs:

- **box** — turns an expression into a code value;
- **unbox** — marks a hole in a code value that can be filled by another code value;
- **run** — executes a code value as code.

For example:

$$\begin{array}{l}\textbf{let}\ y = \textbf{box}\ x\ \textbf{in}\\ \textbf{let}\ z = \textbf{box}\,(1 + (\textbf{unbox}\ y))\ \textbf{in}\\ \textbf{let}\ x = 1\ \textbf{in}\\ \textbf{run}\ z\end{array}$$

Staged Metaprogramming in SLamJS

SLamJS allows staged metaprogramming with these constructs:

- **box** — turns an expression into a code value;
- **unbox** — marks a hole in a code value that can be filled by another code value;
- **run** — executes a code value as code.

For example:

$$\begin{array}{lll}
\textbf{let } y = \textbf{box } x \textbf{ in} & & \\
\textbf{let } z = \textbf{box } (1 + (\textbf{unbox } y)) \textbf{ in} & & \textbf{let } z = \textbf{box } (1 + (\textbf{unbox}(\textbf{box } x))) \textbf{ in} \\
\textbf{let } x = 1 \textbf{ in} & & \textbf{let } x = 1 \textbf{ in} \\
\textbf{run } z & \rightarrow & \textbf{run } z
\end{array}$$

Staged Metaprogramming in SLamJS

SLamJS allows staged metaprogramming with these constructs:

- **box** — turns an expression into a code value;
- **unbox** — marks a hole in a code value that can be filled by another code value;
- **run** — executes a code value as code.

For example:

$$\begin{array}{ll} & \textbf{let } y = \textbf{box } x \textbf{ in} \\ & \textbf{let } z = \textbf{box}\,(1 + (\textbf{unbox } y))\,\textbf{in} \\ & \textbf{let } x = 1\,\textbf{in} \\ & \textbf{run } z \end{array}$$

$$\begin{array}{ll} & \textbf{let } z = \textbf{box}\,(1 + (\textbf{unbox}(\textbf{box}\,x)))\,\textbf{in} \\ & \textbf{let } x = 1\,\textbf{in} \\ \rightarrow & \textbf{run } z \end{array}$$

$$\begin{array}{ll} & \textbf{let } z = \textbf{box}\,(1 + x)\,\textbf{in} \\ & \textbf{let } x = 1\,\textbf{in} \\ \rightarrow & \textbf{run } z \end{array}$$

Staged Metaprogramming in SLamJS

SLamJS allows staged metaprogramming with these constructs:

- **box** — turns an expression into a code value;
- **unbox** — marks a hole in a code value that can be filled by another code value;
- **run** — executes a code value as code.

For example:

$$\begin{array}{l} \textbf{let } y = \textbf{box } x \textbf{ in} \\ \textbf{let } z = \textbf{box}\,(1 + (\textbf{unbox } y))\textbf{ in} \\ \textbf{let } x = 1 \textbf{ in} \\ \textbf{run } z \end{array}
\qquad
\begin{array}{ll} & \textbf{let } z = \textbf{box}\,(1 + (\textbf{unbox}(\textbf{box}\, x)))\textbf{ in} \\ & \textbf{let } x = 1 \textbf{ in} \\ \rightarrow & \textbf{run } z \end{array}$$

$$\begin{array}{ll} & \textbf{let } z = \textbf{box}\,(1 + x)\textbf{ in} \\ & \textbf{let } x = 1 \textbf{ in} \\ \rightarrow & \textbf{run } z \end{array}
\qquad
\begin{array}{ll} & \\ & \textbf{let } x = 1 \textbf{ in} \\ \rightarrow & \textbf{run}\,(\textbf{box}\,(1 + x)) \end{array}$$

Staged Metaprogramming in SLamJS

SLamJS allows staged metaprogramming with these constructs:

- **box** — turns an expression into a code value;
- **unbox** — marks a hole in a code value that can be filled by another code value;
- **run** — executes a code value as code.

For example:

$$\begin{array}{l} \textbf{let } y = \textbf{box } x \textbf{ in} \\ \textbf{let } z = \textbf{box } (1 + (\textbf{unbox } y)) \textbf{ in} \\ \textbf{let } x = 1 \textbf{ in} \\ \textbf{run } z \end{array}$$

$$\begin{array}{l} \textbf{let } z = \textbf{box } (1 + (\textbf{unbox}(\textbf{box } x))) \textbf{ in} \\ \textbf{let } x = 1 \textbf{ in} \\ \rightarrow \textbf{run } z \end{array}$$

$$\begin{array}{l} \textbf{let } z = \textbf{box } (1 + x) \textbf{ in} \\ \textbf{let } x = 1 \textbf{ in} \\ \rightarrow \textbf{run } z \end{array}$$

$$\begin{array}{l} \textbf{let } x = 1 \textbf{ in} \\ \rightarrow \textbf{run } (\textbf{box } (1 + x)) \end{array}$$

$$\begin{array}{l} \textbf{let } x = 1 \textbf{ in} \\ \rightarrow 1 + x \quad \rightarrow \quad 1 + 1 \quad \rightarrow \quad 2 \end{array}$$

Static and dynamic scoping.
No α-equivalence.

Information Flow Analysis for SLamJS

Our information flow analysis for SLamJS comprises two phases:

1. We perform CFA to determine which functions and code values can be bound where.
2. We generate and solve information flow constraints using the results of the CFA.

- Handling code values in CFA requires some special treatment.
- The key observation in CFA is that data and control flow influence each other, so both must be handled in a single analysis. As information flow does not affect data or control flow, it can be separate.
- Because our analysis extends CFA, we believe our technique could easily be adapted to other CFA-style analyses.

CFA for SLamJS

0CFA is a standard analysis that operates by:

1. labelling each subexpression of a program;
2. generating constraints between the values occurring at each label (and each variable);
3. solving these constraints.

0CFA conflates variables with the same name bound in different functions.

- For most languages, this is not a problem, as we can simply α-convert them.
- SLamJS does not respect α-equivalence, so the analysis must track explicitly where names are bound.

CFA for SLamJS

0CFA can be derived from abstract interpretation over a suitable domain. Our abstract domain is:

$$\begin{array}{llcl} \text{Abstract values} & \nu \in \mathit{AbsVal} & ::= & \mathtt{NULL} \mid \mathtt{UNDEF} \mid \mathtt{BOOL} \mid \mathtt{NUM} \mid \mathtt{STR} \\ & & \mid & \mathtt{FUN}(x, e) \mid \mathtt{BOX}(e) \mid \mathtt{REC}(\ell) \end{array}$$

The abstract value $\mathtt{BOX}(e)$ is inhabited by:

- the expression **box** e;
- any expression that **box** e evaluates to.

The range of code values in a program may be infinite. This permissive definition of $\mathtt{BOX}(e)$ ensures that a finite solution to the constraints is always possible.

CFA Example

Consider:

$$((\mathbf{fun}(x)\{\text{l} : (\mathbf{fun}(y)\{x\})\})(\text{H} : 1))(\text{L} : 2) \longrightarrow^{*} 1$$

labelled as:

$$(((\mathbf{fun}(x)\{(\text{l} : (\mathbf{fun}(y)\{x^0\})^1)^2\})^3(\text{H} : 1^4)^5)^6(\text{L} : 2^7)^8)^9$$

Solution of the CFA constraints gives:

$0 \mapsto \{\texttt{NUM}\}$ $\quad 1 \mapsto \{\texttt{FUN}(y, (x)^0)\}$ $\quad 2 \mapsto \{\texttt{FUN}(y, (x)^0)\}$

$3 \mapsto \{\texttt{FUN}(x, ((\text{l} : (\mathbf{fun}(y)\{(x)^0\})^1)^2)\}$ $\quad 4 \mapsto \{\texttt{NUM}\}$ $\quad 5 \mapsto \{\texttt{NUM}\}$

$6 \mapsto \{\texttt{FUN}(y, (x)^0)\}$ $\quad 7 \mapsto \{\texttt{NUM}\}$ $\quad 8 \mapsto \{\texttt{NUM}\}$ $\quad 9 \mapsto \{\texttt{NUM}\}$

$x \mapsto \{\texttt{NUM}\}$ $\quad y \mapsto \{\texttt{NUM}\}$

As expected, the result of evaluation (labelled 9) is a number.

- For a staged example, see our paper.

Information Flow for SLamJS

The information flow anlaysis uses the results of CFA to generate constraints on two relations between markers, labelled program points and variables:

- $\longrightarrow$ tracks direct flows;
- $\rightsquigarrow$ tracks indirect flows.

If an expression marked by $\mathfrak{m}$ is used in computing an expression labelled ℓ then, taking $\leadsto = \longrightarrow \cup \rightsquigarrow$, the analysis ensures $\mathfrak{m} \leadsto^* \ell$.

Theorem (Information Flow Soundness)

Suppose $\leadsto$ has been computed for t^ℓ by the information flow analysis. Consider $t^\ell \longrightarrow^* v^{\ell'}$, where v is an unstaged value composed only of markers and constants, and a marker $\mathfrak{m}$ with $\mathfrak{m} \not\leadsto^* \ell$. Now let t' be t with a subexpression marked with $\mathfrak{m}$ replaced by some other expression. Then $t'^\ell \longrightarrow^* v^{\ell'}$.

The key parts of this theorem have been mechanised in Coq.

Information Flow Analysis Example

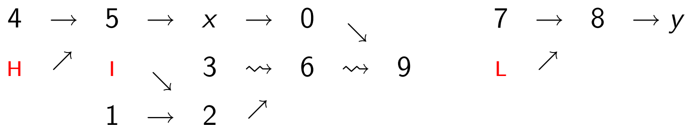
Recall:

$$((\mathbf{fun}(x)\{\mathsf{I} : (\mathbf{fun}(y)\{x\})\})(\mathsf{H} : 1))(\mathsf{L} : 2) \longrightarrow^{*} 1$$

labelled as:

$$(((\mathbf{fun}(x)\{(\mathsf{I} : (\mathbf{fun}(y)\{x^{0}\})^{1})^{2}\})^{3}(\mathsf{H} : 1^{4})^{5})^{6}(\mathsf{L} : 2^{7})^{8})^{9}$$

The information flow constraints are:



We have $\mathsf{H} \rightsquigarrow^{*} 9$ and $\mathsf{I} \rightsquigarrow^{*} 9$ and $\mathsf{L} \not\rightsquigarrow^{*} 9$. This means the result (labelled 9) has information flows from H and I, but not L.

Future Work

- Extend the analysis to handle other JavaScript features, such as mutable state and exceptions.
- Improve the precision of analysis of object reads and writes by extending the abstract string domain.
- Transfer our ideas to a CFA2 analysis for improved precision with higher order flow.
- Apply recent work on analysing **eval** directly to transform uses of **eval** into staged metaprogramming.

Conclusion

Our contributions:

- We have developed an information flow analysis for a JavaScript-like language with staged metaprogramming.
- We have mechanised the proof of soundness for our analysis using Coq.
- We have implemented our analysis in OCaml.
- Online material: `http://mjolnir.cs.ox.ac.uk/web/slamjs/`.

We believe that we now have all the technical tools for an interesting information flow analysis of JavaScript with **eval**.

Conclusion

Our contributions:

- We have developed an information flow analysis for a JavaScript-like language with staged metaprogramming.
- We have mechanised the proof of soundness for our analysis using Coq.
- We have implemented our analysis in OCaml.
- Online material: `http://mjolnir.cs.ox.ac.uk/web/slamjs/`.

We believe that we now have all the technical tools for an interesting information flow analysis of JavaScript with **eval**.

- Thanks for listening. Questions are welcome.