Information Flow Analysis for a Dynamically Typed Functional 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

LOng Lunch Seminar, 2012–10–30

Information Flow Analysis for JavaScript

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

[1]Department of Computer Science, University of Oxford

[2]IBM T.J. Watson Research Center

LOng Lunch Seminar, 2012–10–30

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**.

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.

Uses and Abuses of Eval

Many research papers claim that **eval** is used rarely or only in trivial ways. A recent survey shows otherwise. Examples include:

- concatenating strings to form variable names;
- simulating higher order functions;
- bizarre or seemingly pointless invocations.

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

$$\begin{array}{rc} \text{Plug:} & \langle x \rangle \\ \text{into:} & (\mathbf{fun}(\langle - \rangle)\{x+1\}) \\ \text{to get:} & (\mathbf{fun}(x)\{x+1\}) \end{array}$$

✓

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

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

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.

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.

Verifying Noninterference

Some early work on noninterference added a monitor to a program to track and enforce security levels of variables. Unfortunately:

- A 1-safety property of a program is one for which a violation can be shown by a single, finite trace of a program.
- Monitoring is good for enforcing 1-safety properties.
- Showing violation of a 2-safety property requires two traces.
- Noninterference is a 2-safety property.

Some recent research revisits monitoring and works around this by combining it with simple static analysis.

- Monitoring is good at handling metaprogramming.
- But it is still a dynamic (rather than static) analysis.

There is a large body of research on verifying noninterference with type systems, for example in ML.

- This is difficult to apply to dynamically typed languages.

Information Flow Analysis

- 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).
- "x is not used to compute y" is a stronger claim than "the value of x does not affect the value of y".

Example

This program clearly uses x in its computation of y, but y is always 0, so x does not affect its value.

$$
\begin{array}{l}
x := 10; \\
\textbf{while}(x <> 0)\{ \\
\quad x := x - 1; \\
\} \\
y := x;
\end{array}
$$

- We can use an information flow analysis to verify noninterference.

Outline

Syntax of SLamJS

$$
\begin{array}{lllll}
\text{Booleans} & b & ::= & \textbf{true} \mid \textbf{false} \\
\text{Strings} & s & \in & \mathit{String} \\
\text{Numbers} & n & \in & \mathit{Number} \\
\text{Names} & x & \in & \mathit{Name} \\
\text{Constants} & k & ::= & \textbf{undef} \mid \textbf{null} \mid b \mid s \mid n \\
\text{Expressions} & e & ::= & k \mid \{\overline{s : e}\} \mid x \mid \textbf{fun}(x)\{e\} \mid e(e) \mid \textbf{box}\ e \\
 & & & \mid \textbf{unbox}\ e \mid \textbf{run}\ e \mid \textbf{if}(e)\{e\}\,\textbf{else}\{e\} \mid e[e] \\
 & & & \mid e[e] = e \mid \textbf{del}\ e[e] \mid (e, \rho) \mid \textbf{run}\ e\ \textbf{in}\ \rho \\
\text{Values} & v, v^0 & ::= & (\textbf{fun}(x)\{e\}, \rho) \\
 & v^n & ::= & k \mid \{\overline{s : v^n}\} \mid (\textbf{box}\ v^{n+1}) \\
 & v^{n+1} & ::= & x \mid (\textbf{fun}(x)\{v^{n+1}\}) \mid (v^{n+1}(v^{n+1})) \\
 & \multicolumn{3}{l}{\mid (\textbf{run}\ v^{n+1}) \mid (\textbf{if}(v^{n+1})\{v^{n+1}\}\,\textbf{else}\{v^{n+1}\})} \\
 & \multicolumn{3}{l}{\mid (v^{n+1}[v^{n+1}]) \mid (v^{n+1}[v^{n+1}] = v^{n+1}) \mid (\textbf{del}\ v^{n+1}[v^{n+1}])} \\
 & v^{n+2} & ::= & (\textbf{unbox}\ v^{n+1}) \\
\text{Environments} & \rho & \in & \mathit{Name} \xrightarrow{\text{fin}} v^0
\end{array}
$$

Syntax of SLamJS

Booleans	b	$::=$	$\textbf{true} \mid \textbf{false}$
Strings	s	$\in$	*String*
Numbers	n	$\in$	*Number*
Names	x	$\in$	*Name*
Constants	k	$::=$	$\textbf{undef} \mid \textbf{null} \mid b \mid s \mid n$
Expressions	e	$::=$	$k \mid x \mid \textbf{fun}(x)\{e\} \mid e(e) \mid \textbf{box}\ e$
			$\mid \textbf{unbox}\ e \mid \textbf{run}\ e \mid \textbf{if}(e)\{e\}\,\textbf{else}\{e\}$
Values	v, v^0	$::=$	$(\textbf{fun}(x)\{e\}, \rho)$
	v^n	$::=$	$k \mid (\textbf{box}\ v^{n+1})$
	v^{n+1}	$::=$	$x \mid (\textbf{fun}(x)\{v^{n+1}\}) \mid (v^{n+1}(v^{n+1}))$
	$\mid (\textbf{run}\ v^{n+1}) \mid (\textbf{if}(v^{n+1})\{v^{n+1}\}\,\textbf{else}\{v^{n+1}\})$		
	v^{n+2}	$::=$	$(\textbf{unbox}\ v^{n+1})$

- We ignore the prototype-based objects in the rest of this presentation.
- Environments ρ occur only during evaluation.

Semantics of SLamJS

We define evaluation contexts and an evaluation relation $\xrightarrow{\boxplus}$ in a typical way. Here are some illustrations of its behaviour:

- $(\mathbf{fun}(x)\{x\})(1) \xrightarrow{\boxplus} 1$
- $((\mathbf{fun}(x)\{\mathbf{fun}(y)\{x\}\})(1))(2) \xrightarrow{\boxplus} 1$
- $\mathbf{if}(\mathbf{true})\{1\}\,\mathbf{else}\{\mathbf{false}\} \xrightarrow{\boxplus} 1$
- $(\mathbf{fun}(x)\{\mathbf{run}(\mathbf{box}\,x)\})(0) \xrightarrow{\boxplus} 0$
- $\mathbf{run}(\mathbf{box}(\mathbf{if}(\mathbf{unbox}(\mathbf{box}\,\mathbf{true}))\{1\}\,\mathbf{else}\{\mathbf{false}\})) \xrightarrow{\boxplus} 1$

Information Flow in SLamJS

To allow us to express information flow in SLamJS, we augment it with explicit security level markers:

$$\begin{array}{llll} \text{Markers} & \mathfrak{m} & \in & \mathit{Marker} \\ \text{Expressions} & e & ::= & \ldots \mid (\mathfrak{m} : e) \end{array}$$

Direct flows from a marked value are tracked by the marker being part of the value:

- $\mathbf{if}(\mathbf{false})\{\mathbf{true}\}\,\mathbf{else}\{\mathsf{L} : l\} \xrightarrow{\square} \mathsf{L} : l$

No further treatment is needed.
Markers block reductions that might result in an indirect flow:

- $\mathbf{if}(\mathsf{H} : h)\{\mathbf{true}\}\,\mathbf{else}\{\mathsf{L} : l\} \not\xrightarrow{\square}$

We introduce lift rules that move markers towards the top level of an expression:

- $\mathbf{if}(\mathsf{H} : h)\{\mathbf{true}\}\,\mathbf{else}\{\mathsf{L} : l\} \xrightarrow{\square} \mathsf{H} : (\mathbf{if}(h)\{\mathbf{true}\}\,\mathbf{else}\{\mathsf{L} : l\})$

Effectively, an indirect flow is turned into a direct one.

Erasure in SLamJS

Erasure captures what it means for a value *not* to be used in a computation.

- The M-erasure of e, written $\lfloor e \rfloor_M$, is e with all subexpressions marked by $\mathfrak{m} \notin M$ replaced with _.
- _ behaves like an unbound variable.
- $\lfloor \mathbf{if}(\mathbf{true})\{\mathbf{false}\}\,\mathbf{else}\{\mathsf{H} : h\} \rfloor_{\mathsf{L}} = \mathbf{if}(\mathbf{true})\{\mathbf{false}\}\,\mathbf{else}\{\_\} \xrightarrow{\boxminus} \mathbf{false}$
- $\lfloor \mathbf{if}(\mathsf{H} : h)\{\mathbf{true}\}\,\mathbf{else}\{\mathsf{L} : l\} \rfloor_{\mathsf{L}} = \mathbf{if}(\_)\{\mathbf{true}\}\,\mathbf{else}\{\mathsf{L} : l\} \not\xrightarrow{\boxminus}$

Theorem (Stability)

Consider an expression e_1 (which may use _) and a _-free expression e_2 such that $e_1 \xrightarrow{\boxminus}{}^* e_2$. Then for every $M \subseteq Marker$ such that $\lfloor e_2 \rfloor_M = e_2$, it follows that $\lfloor e_1 \rfloor_M \xrightarrow{\boxminus}{}^* \lfloor e_2 \rfloor_M$.

- This means that if e_2 is not marked by $\mathfrak{m}$, we can safely erase it from e_1.

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:

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

The abstract value $\texttt{BOX}(e)$ 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 $\texttt{BOX}(e)$ ensures that a finite solution to the constraints is always possible.

Sample CFA rules

$$\Gamma, \varrho \models k^{\ell} \quad \text{if} \quad \lceil k \rceil \in \Gamma(\ell)$$

$$\Gamma, \varrho \models x^{\ell} \quad \text{if} \quad \varrho(x) \subseteq \Gamma(\ell)$$

$$\Gamma, \varrho \models (\textbf{box}\ e)^{\ell} \quad \text{if} \quad \Gamma, \varrho \models e \quad \text{and} \quad \exists \nu \in \Gamma(\ell).\Gamma, \varrho \models \nu \approx \textbf{box}\ e$$

$$\Gamma, \varrho \models (\textbf{unbox}\ e)^{\ell} \quad \text{if} \quad \Gamma, \varrho \models e \quad \text{and} \quad \forall \mathtt{BOX}(e') \in \Gamma(\mathit{lbl}(e)).\Gamma(\mathit{lbl}(e')) \subseteq \Gamma(\ell)$$

$$\Gamma, \varrho \models (\textbf{if}(e_1)\{e_2\}\ \textbf{else}\{e_3\})^{\ell} \quad \text{if} \quad \Gamma, \varrho \models e_1 \wedge \Gamma, \varrho \models e_2 \wedge \Gamma, \varrho \models e_3 \quad \text{and} \quad \Gamma(\mathit{lbl}(e_2)) \subseteq \Gamma(\ell) \wedge \Gamma(\mathit{lbl}(e_3)) \subseteq \Gamma(\ell)$$

CFA Example

Consider:

$$(((\textbf{fun}(x)\{\mathsf{I} : (\textbf{fun}(y)\{x\})\})(\mathsf{H} : 1))(\mathsf{L} : 2), \epsilon) \xrightarrow{\square}{}^{*} (\mathsf{I} : (\mathsf{H} : 1))$$

labelled as:

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

Solution of the CFA constraints gives:

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

$3 \mapsto \{\mathtt{FUN}(x, ((\mathsf{I} : (\textbf{fun}(y)\{(x)^0\})^1)^2)\}$ $\quad$ $4 \mapsto \{\mathtt{NUM}\}$ $\quad$ $5 \mapsto \{\mathtt{NUM}\}$

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

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

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

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:

- $\rightsquigarrow$ tracks direct flows;
- $\not\rightsquigarrow$ tracks indirect flows.

If an expression marked by $\mathfrak{m}$ is used in computing an expression labelled ℓ then, taking $\rightsquigarrow = \rightsquigarrow \cup \not\rightsquigarrow$, the analysis ensures $\mathfrak{m} \rightsquigarrow^* \ell$.

Theorem (Information Flow Soundness)

Suppose $\rightsquigarrow$ *has been computed for* t^ℓ *by the information flow analysis. Then if* $t^\ell \xrightarrow{\not\square}{}^* v^{\ell'}$, *where* v *is a stage-0 value composed only of markers and constants, and* $M = \{\mathfrak{m} \in \mathit{Marker} \mid \mathfrak{m} \rightsquigarrow^* \ell\}$, *it follows that* $\lfloor v \rfloor_M = v$.

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

Sample Information Flow Rules

Expression e $\models_{\mathrm{IF}} e$ holds:	*Subexpressions* if:	*Direct* and:	*Indirect* and:
k^{ℓ}	–	–	–
x^{ℓ}	–	$x \rightsquigarrow \ell$	–
$(\textbf{if}(t_1^{\ell_1})\{t_2^{\ell_2}\}\,\textbf{else}\{t_3^{\ell_3}\})^{\ell_4}$	$\bigwedge_{i=1}^{3} \models_{\mathrm{IF}} t_i^{\ell_i}$	$\ell_2 \rightsquigarrow \ell_4 \wedge \ell_3 \rightsquigarrow \ell_4$	$\ell_1 \looparrowright \ell_4$
$(\textbf{box}\ t^{\ell_1})^{\ell_2}$	$\models_{\mathrm{IF}} t^{\ell_1}$	–	–
$(\textbf{unbox}\ t^{\ell_1})^{\ell_2}$	$\models_{\mathrm{IF}} t^{\ell_1}$	$\forall \mathtt{BOX}(t'^{\ell'}) \in \Gamma(\ell_1)\,.$ $\ell' \rightsquigarrow \ell_2$	$\ell_1 \looparrowright \ell_2$

Information Flow Analysis Example

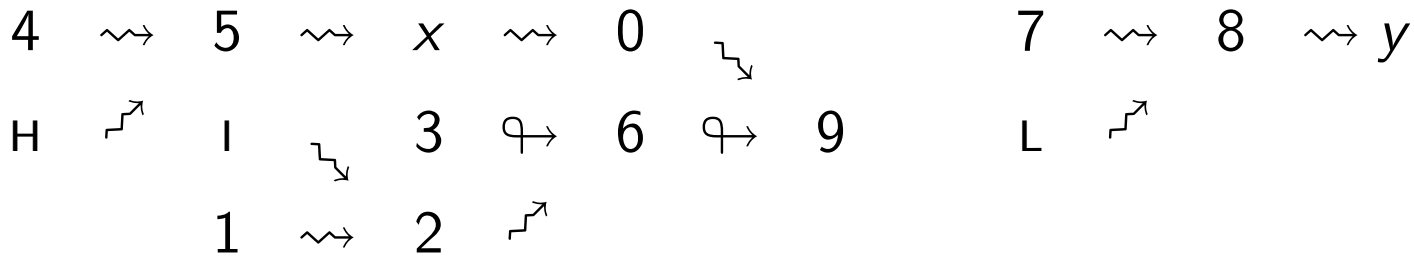
Recall:

$$(((\mathbf{fun}(x)\{\mathsf{I} : (\mathbf{fun}(y)\{x\})\})(\mathsf{H} : 1))(\mathsf{L} : 2), \epsilon) \xrightarrow{\square}{}^{*} (\mathsf{I} : (\mathsf{H} : 1))$$

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.

Impelementation

We have implemented our analysis in OCaml.

Example

$$
\begin{array}{l}
\textbf{let } c = \textbf{box } x \textbf{ in} \\
\textbf{let } x = \mathsf{L} : 1 \textbf{ in} \\
\textbf{let } \mathit{eval} = \textbf{fun}(b)\{\textbf{run } b\} \textbf{ in} \\
\textbf{let } x = \mathsf{H} : 2 \textbf{ in} \\
\mathit{eval}(c)
\end{array}
$$

Depends on: L

Impelementation

We have implemented our analysis in OCaml.

Example

$$
\begin{array}{l}
\textbf{let } c = \textbf{box } x \textbf{ in} \\
\textbf{let } x = \mathsf{L} : 1 \textbf{ in} \\
\textbf{let } \mathit{eval} = \textbf{fun}(b)\{\textbf{run } b\} \textbf{ in} \\
\textbf{let } x = \mathsf{H} : 2 \textbf{ in} \\
\mathit{eval}(c)
\end{array}
$$

Depends on: L

$$
\begin{array}{l}
\textbf{let } x = \textbf{if}(\textbf{true})\{\textbf{box } f\} \textbf{ else}\{\textbf{box } g\} \textbf{ in} \\
\textbf{let } f = \textbf{fun}(y)\{1\} \textbf{ in} \\
\textbf{let } g = \textbf{fun}(z)\{\mathsf{L} : \textbf{true}\} \textbf{ in} \\
\textbf{run } (\textbf{box } ((\textbf{unbox } x)(\mathsf{H} : \textbf{undef})))
\end{array}
$$

Depends on: 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.