Information Flow Analysis for JavaScript
via a dynamically typed language with staged metaprogramming

Martin Lester

Department of Computer Science, Parks Road, Oxford, OX1 3QD, UK
`http://mjolnir.cs.ox.ac.uk/web/slamjs/`

Web applications written in JavaScript are regularly used for dealing with sensitive or personal data. Consequently, reasoning about their security properties has become an important problem, which is made very difficult by the highly dynamic nature of the language, particularly its support for *runtime code generation* through the **eval** construct.

We have developed a *static information flow* analysis for a dynamically typed, functional language with *staged metaprogramming* that has semantics similar to JavaScript. Our analysis works by extending the well-known Control Flow Analysis (CFA) with an abstraction for staged code and information flow constraints. We have formally proved the soundness of our analysis in the mechanised theorem prover Coq and implemented it in OCaml [6].

This is the first information flow analysis for a language with staged metaprogramming and the first formal soundness proof of a CFA-based information flow analysis for a functional language. We argue that our work is applicable to an information flow analysis for full JavaScript, including **eval**, and could easily be transferred to other CFA-based analyses.

Information Flow and Noninterference The study of *information flow* security stems from the observation that programs contain both *direct* and *indirect* information flows [2].

Consider evaluation of the program ($\mathsf{if}(h)\{l\}\,\mathsf{else}\{0\}$). The result may be l, so there is a *direct* flow from l to the result. However, we might also infer from the result whether h is **true** or **false**. As h affects the result of the program through control flow, but does not directly contribute to the result, there is an *indirect* flow from h to the result.

Early work on information flow focused on augmenting program execution with a monitor to *taint* variables that contained high-security data [3]. But this method is weak at handling indirect flows, which may arise from program branches not executed in all runs.

There is a wide body of research concerning information flow analysis and security in statically typed languages [7], but relatively little for dynamically typed languages.

A popular information flow security property is *noninterference* [4]. Suppose that the inputs and outputs to a program are partitioned into different security levels. For example, some may be high-security (or *high* for short) while others are low-security (or *low*). A program satisfies noninterference if its high inputs do not affect its low outputs.

In the context of a Web application, a high input might be a text input box for a credit card number and a low output might be an unencrypted connection to a webserver. Such an application would satisfy a noninterference analysis if unencrypted transmissions could never reveal anything about the credit card number. Our analysis identifies which program values (inputs) might affect the result (output), so it can be used to verify noninterference.

Staged Metaprogramming JavaScript's **eval** construct takes a string and executes it as if it were a piece of program code. This is difficult to analyse for three reasons. Firstly, an analysis must determine what code strings may be passed to the **eval** and what code they may represent. Secondly, there may be infinitely many possible code strings, so there may be infinitely many subprograms to analyse. Thirdly, code executed by **eval** operates under different scoping rules. Not only must the analysis handle a mixture of *dynamic and static variable scoping*, but code using **eval** does not respect α-equivalence.

Metaprogramming constructs are poorly understood, but use of code strings makes **eval** particularly tricky. As a step towards a more principled analysis, we use a language with Lisp-style staged metaprogramming: programs can construct (**box**), splice together (**unbox**) and execute (**run**) code templates, but code values will always be syntactically valid programs.

Analysis In order to express which values are high, low and intermediate, we extend our language with security markers H, L, I, …. They have no computational role; they simply indicate the security level of a value.

Our analysis is based on the popular dataflow analysis CFA [8]. CFA assigns each subexpression of a program a distinct label to track which values it may evaluate to. It generates constraints between labels to express *control and data flow* within the program.

First we extend the analysis to handle staged metaprogramming. Applying an idea from recent work [1], we model code values in a similar way to functions, but with different scoping rules: code values use the scope where they are run, not defined. Next we add constraints to track direct and indirect *information flows*. Here is an example without staging:

$$((\textbf{fun}(x)\{\textsc{i} : (\textbf{fun}(y)\{x\})\})(\textsc{h} : 1))(\textsc{l} : 2) \to^* 1$$

with labels on each subexpression:

$$(((\textbf{fun}(x)\{(\textsc{i} : (\textbf{fun}(y)\{x^0\})^1)^2\})^3(\textsc{h} : 1^4)^5)^6(\textsc{l} : 2^7)^8)^9$$

$$\begin{tikzcd}
4 \arrow[r] & 5 \arrow[r] & x \arrow[r] & 0 \arrow[dr] & & 7 \arrow[r] & 8 \arrow[r] & y \\
\textsc{h} \arrow[ur] & \textsc{i} \arrow[dr] & 3 \arrow[r, squiggly] & 6 \arrow[r, squiggly] & 9 & \textsc{l} \arrow[ur] & & \\
& 1 \arrow[r] & 2 \arrow[ur] & & & & &
\end{tikzcd}$$

The information flow constraints, direct ($\rightarrow$) and indirect ($\rightsquigarrow$), are shown on the right. The result (labelled 9) depends directly on H, indirectly on I and not on L. The indirect flows result from applying functions, which can reveal the identity of the applied function.

Next we have an example with staging:

$$\begin{aligned}
&(\textbf{let}\ x = (\textsc{l} : 1^0)^1\ \textbf{in}\\
&(\textbf{let}\ c = (\textbf{box}\, x^2)^3\ \textbf{in}\\
&(\textbf{let}\ x = (\textsc{h} : 2^4)^5\ \textbf{in}\\
&(\textbf{run}\ c^6)^7)^8)^9)^{10} \to^* 2
\end{aligned}$$

$$\begin{tikzcd}
4 \arrow[r] & 5 \arrow[r] & x@7 \arrow[r, leftrightarrow] & x@2 \arrow[r] & 2 \arrow[r] & 7 \arrow[r] & 8 \arrow[r] & 9 \arrow[r] & 10 \\
\textsc{h} \arrow[ur] & & c@2 \arrow[r, leftrightarrow] & c@8 \arrow[r] & 6 \arrow[ur, dashed] & & \textsc{l} \arrow[dr] & & \\
& & 3 \arrow[ur] & & & & 0 \arrow[r] & 1 \arrow[r] & x@9
\end{tikzcd}$$

Here **box** x is defined in the scope of the first **let** $x = \ldots$, which is labelled 9; the label $x@9$ tracks the values of x in this scope. But **box** x is run in (and so uses) the scope of the second **let** $x = \ldots$ ($x@7$). Thus the result (labelled 10) depends on H, but not L.

Future Work We believe all the pieces are now in place for an interesting, principled analysis of JavaScript with **eval**, but it will take a significant effort to combine them. Our analysis only handles a subset of JavaScript's features and is quite coarse in its abstraction of basic datatypes, but our ideas could be applied to a state-of-the-art CFA-based analysis. We also need to show how to transform string-based **eval** soundly into staged metaprogramming; recent research on direct analysis of **eval** suggests a way of doing that [5]. Finally, there are many infrastructural issues concerning how such an analysis would be used in practice.

References

1. Choi, W., Aktemur, B., Yi, K., and Tatsuta, M. Static Analysis of Multi-staged Programs via Unstaging Translation. In *POPL* (2011), pp. 81–92.
2. Denning, D. E. A Lattice Model of Secure Information Flow. *CACM 19*, 5 (1976), 236–243.
3. Fenton, J. S. Memoryless subsystems. *Comput. J. 17*, 2 (1974), 143–147.
4. Goguen, J. A., and Meseguer, J. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy* (1982), pp. 11–20.
5. Jensen, S. H., Jonsson, P. A., and Møller, A. Remedying the Eval that Men Do. In *ISSTA* (2012), pp. 34–44.
6. Lester, M., Ong, L., and Schäefer, M. Information Flow Analysis for a Dynamically Typed Language with Staged Metaprogramming. Submitted to POST 2013.
7. Pottier, F., and Simonet, V. Information Flow Inference for ML. *TOPLAS 25*, 1 (2003).
8. Shivers, O. Control-Flow Analysis in Scheme. In *PLDI* (1988), pp. 164–174.