Hybrid Typing Secure Information Flow in a Core of JavaScript

José Fragoso Santos

December 17, 2014

Problem: Property names are computed dynamically

Example:

```
o = { };
o.secret = secret_input();
o.public1 = public_input();
o.public2 = public_input();
public_out = o[f()]
```

Remarks:

- When `f()` evaluates to `"secret"` $\Rightarrow$ **illegal flow**
- When `f()` does **NOT** evaluate to `"secret"` $\Rightarrow$ **only legal flow**

Idea: Combine Typing with Rewritting

Idea

Use a type-directed transformation to cut illegal behaviors.

Original Program:

```
o = { };
o.secret = secret_input();
o.public1 = public_input();
o.public2 = public_input();
public_out = o[f()]
```

Transformed Program:

```
o = { };
o.secret = secret_input();
o.public1 = public_input();
o.public2 = public_input();
_x = f();
if (_x !== "secret") {
public_out = o[f()]
} else {
abort()
}
```

Idea: Combine Typing with Rewritting

Idea

Use a type-directed transformation to cut illegal behaviors.

Problem

How to automate this type of transformation?

Core JavaScript

Defining Features of the Language

1. Extensible Objects
2. Prototype-based Inheritance
3. Functions as first class values
4. Closures
5. Constructs for checking the existence of object properties
6. Atypical interactions between the binding of properties and the binding of variables

Core JavaScript

Syntax

$e \in \texttt{Expr}$	$::=$	v	% Value
	$\mid$	$\textsf{this}^i$	% This
	$\mid$	x^i	% Identifier
	$\mid$	$e_0\ \textsf{op}^i\ e_1$	% Binary operation
	$\mid$	$x = e$	% Variable Assignment
	$\mid$	$e_0[e_1]^i$	% Property Look-up
	$\mid$	$e_0\ \textsf{in}^i\ e_1$	% Membership Testing
	$\mid$	$e_0[e_1] = e_2$	% Property Assignment
	$\mid$	$\textsf{delete}^i\ e_0[e_1]$	% Property Deletion
	$\mid$	$e_0(e_1)^i$	% Function Call
	$\mid$	$e_0[e_1](e_2)^i$	% Method Call
	$\mid$	$e_0\ ?^{i,j}\ (e_1) : (e_2)$	% Conditional
	$\mid$	$e_0,\ e_1$	% Sequence
	$\mid$	$\{\,\}^i$	% Object Literal
	$\mid$	$\textsf{function}^i(x)\{\textsf{var}\ y_1, \cdots, y_n;\ e\}$	% Function Literal

Information Flow Security in One Slide

Idea

Public Outputs (LOW) may NOT depend on Private Inputs (HIGH)

Ingredients

1. A **lattice of security levels**
2. A **security labelling** mapping resources to security types

Information Flow Types for a Core of JavaScript

Idea

Annotate safety types with security levels

Syntax of Security Types

$$
\begin{array}{rcll}
\tau \in \texttt{Type} & ::= & \textsf{PRIM} & \text{\% Prim Type} \\
 & & \mid\ \langle \dot{\tau}.\dot{\tau} \stackrel{\sigma}{\to} \dot{\tau} \rangle & \text{\% Function Type} \\
 & & \mid\ \langle \kappa.\dot{\tau} \stackrel{\sigma}{\to} \dot{\tau} \rangle & \text{\% Method Type} \\
 & & \mid\ \mu\kappa.\langle p^{\sigma} : \dot{\tau}, \cdots, p^{\sigma} : \dot{\tau}, *^{\sigma} : \dot{\tau} \rangle & \text{\% Ext Obj Type} \\
 & & \mid \mu\kappa.\langle p^{\sigma} : \dot{\tau}, \cdots, p^{\sigma} : \dot{\tau} \rangle & \text{\% NonExt Obj Type} \\
\dot{\tau} \in \texttt{SType} & ::= & \tau^{\sigma} & \text{\% Security Type}
\end{array}
$$

Attacker Model - 1

Important Questions

1. What can an attacker know about the contents of a JavaScript memory?
2. How can he use the language in order to learn it?

Short Answer:

1. Values of Variables
2. Values of Properties
3. Existence of Properties

Attacker Model - 2

Type-Based Labellings

$$r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu', \Sigma', v \rangle$$

A type based labelling is a mapping from references to their security types.

$\mu \upharpoonright^{\Sigma,\sigma}$ – What can an attacker see at a given level σ?

1. The existence of properties whose existence level is $\leq \sigma$
2. The values associated with properties whose level is $\leq \sigma$
3. The value associated with variables whose level is $\leq \sigma$

Low-Equality for Labelled Memories

$$\mu, \Sigma \sim_\sigma \mu', \Sigma' \quad \text{iff} \quad \mu \upharpoonright^{\Sigma,\sigma} = \mu' \upharpoonright^{\Sigma',\sigma}$$

Noninterference

Consistency

A typing environment Γ must be consistent with the type based labelling Σ. **Example.** Suppose $x \in dom(\Gamma)$ and $\Gamma(x)$ is an object type, then:

$$\Gamma(x) = \Sigma(\mu(r)(x))$$

The expression e is **noninterferent** with respect to Γ iff

for any two memories μ and μ', type-based labellings Σ and Σ', and security level $\sigma \in \mathcal{L}$ such that:

1. Σ and Σ' are **consistent** with Γ,
2. $\#glob \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v \rangle$,
3. $\#glob \vdash \langle \mu', \Sigma', e \rangle \Downarrow \langle \mu'_f, \Sigma'_f, v' \rangle$, and
4. $\mu, \Sigma \sim_\sigma \mu', \Sigma'$;

It holds that: $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$.

Static Typing

Typing Judgements

$$\Gamma, \sigma_{pc} \vdash e : \dot{\tau}$$

1. Γ is the typing environment
2. σ_{pc} the *context level*
3. e is the expression to be typed
4. $\dot{\tau}$ the type that is assigned to it

Hybrid Typing - General Ideas

Ideas

1. Rewrite each expression in order to bookkeep the values of intermediate expressions
2. Type each expression with the set of all its possible types
3. Each type is paired up with a runtime assertion that describes the conditions under which it is applicable
4. Constraints that cannot be verified statically should be verified dynamically

Hybrid Typing

Typing Judgements

$$\Gamma, L_{pc} \vdash e \rightsquigarrow {}^{e'}/_{e''} : T$$

1. Γ is the typing environment
2. L_{pc} is a level set that represents all the possible levels of the current context,
3. e is the expression to be typed
4. e' is a new expression semantically equivalent to e except for the executions that are considered illegal,
5. e'' is an expression that bookkeeps the value to which e' evaluates,
6. T is the type set representing all possible types of e.

Type Sets and Level Sets

Type Sets

A **type set** T is a set of security types paired up with runtime assertions:

$$T = \{(\dot{\tau}_0, \omega_0), \cdots, (\dot{\tau}_n, \omega_n)\}$$

Level Sets

A **level set** L is a set of security types paired up with runtime assertions:

$$L = \{(\sigma_0, \omega_0), \cdots, (\sigma_n, \omega_n)\}$$

A Program Logic for Reasoning about Local Scope

Idea

Add new variables to bookkeep the values of intermediate expressions.

Syntax of Runtime Assertions

$$\omega ::= \$v_i \in V \mid v \in V \mid \mathsf{true} \mid \omega \vee \omega \mid \omega \wedge \omega \mid \neg\omega$$

Satisfaction Relation for Runtime Assertions

$$\begin{array}{rcl}
\mu, r \vDash \$v_i \in V & \Leftrightarrow & r' = \mathsf{Scope}(\mu, r, \$v_i) \ \wedge \ \mu(r' \cdot \mathsf{string}(\$v_i)) \in V \\
\mu, r \vDash \omega_0 \vee \omega_1 & \Leftrightarrow & \mu, r \vDash \omega_0 \ \vee \ \mu, r \vDash \omega_1 \\
\mu, r \vDash \omega_0 \wedge \omega_1 & \Leftrightarrow & \mu, r \vDash \omega_0 \ \wedge \ \mu, r \vDash \omega_1 \\
\mu, r \vDash \neg\omega & \Leftrightarrow & \mu, r \nvDash \omega \\
\mu, r \vDash \mathsf{true} & \Leftrightarrow & \textit{always}
\end{array}$$

Typing Rules - Variable Assignment

Static

$$\frac{\Gamma, \sigma_{pc} \vdash e : \dot{\tau} \qquad \dot{\tau}^{\sigma_{pc}} \preceq \Gamma(x)}{\Gamma, \sigma_{pc} \vdash x = e : \dot{\tau}}$$

Hybrid

$$\frac{\Gamma, L_{pc} \vdash e_0 \rightsquigarrow {}^{e'_0}/_{e''_0} : T_0 \quad \omega = \mathsf{When}^{?}_{\preceq}(T_0^{L_{pc}}, \Gamma(x)) \quad e = e'_0, \mathsf{Wrap}(\omega, x = e''_0)}{\Gamma, L_{pc} \vdash x = e_0 \rightsquigarrow {}^{e}/_{e''_0} : T_0}$$

Operations on Type Sets - 1

The Operator **When**

$$\omega = \text{When}^{?}_{\preceq}(T_0, T_1)$$

ω is the assertion that describes the conditions under which there are two pairs: $(\dot{\tau}_0, \omega_0) \in T_0$ and $(\dot{\tau}_1, \omega_1) \in T_1$ such that $\dot{\tau}_0 \preceq \dot{\tau}_1$ and $\omega_0 \wedge \omega_1$ holds.

Exponentiation with Level Set - T^L

$$T^L = \{(\dot{\tau}', \omega) \mid (\dot{\tau}, \omega_t) \in T \ \wedge \ (\sigma, \omega_l) \in L \ \wedge \ \omega = \omega_t \wedge \omega_l \ \wedge \ \dot{\tau}' = \dot{\tau}^{\sigma}\}$$

Typing Rules - Binary Operation

Static

$$\frac{\Gamma, \sigma_{pc} \vdash e_i : \dot{\tau}_i \quad \dot{\tau} = \dot{\tau}_0 \curlyvee \dot{\tau}_1}{\Gamma, \sigma_{pc} \vdash e_0 \ \mathsf{op}\ e_1 : \dot{\tau}}$$

Hybrid

$$\frac{\forall_{i=0,1} \cdot \Gamma, L_{pc} \vdash e_i \leadsto {}^{e'_i}/_{e''_i} : T_i \quad e' = e'_0, e'_1, \$v_j = e''_0 \ \mathsf{op}\ e''_1}{\Gamma, L_{pc} \vdash e_0 \ \mathsf{op}^j\ e_1 \leadsto {}^{e'}/_{\$v_j} : T_0 \oplus_{\curlyvee} T_1}$$

Operations on Type Sets - 2

Combining Type Sets

$$(\dot{\tau}, \omega) \in T_0 \oplus_{\curlyvee} T_1$$

For every memory μ and reference r, $\mu, r \vDash \omega$ if and only if:

- $(\dot{\tau}_0, \omega_0) \in T_0$
- $(\dot{\tau}_1, \omega_1) \in T_1$
- $\mu, r \vDash (\omega_0 \wedge \omega_1)$
- $\dot{\tau} = \dot{\tau}_0 \curlyvee \dot{\tau}_1$

Typing Rules - Property Lookup

Static

$$\frac{\forall_{i=0,1} \cdot \Gamma, \sigma_{pc} \vdash e_i : \dot{\tau}_i \quad \dot{\tau} = \pi_{\texttt{type}}(\upharpoonright_{\uparrow} (\dot{\tau}_0, P)) \\ \sigma = lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1)}{\Gamma, \sigma_{pc} \vdash e_0[e_1, P] : \dot{\tau}^{\sigma}}$$

Hybrid

$$\frac{\forall_{i=0,1} \cdot \Gamma, L_{pc} \vdash e_i \rightsquigarrow {}^{e'_i}/_{e''_i} : T_i \\ T_P = \pi_{\texttt{type}}(\upharpoonright^{?} (T_0, P, e''_1)) \\ L = lev(T_0) \oplus_{\sqcup} lev(T_1) \quad e = e'_0, e'_1, \$v_j = e''_0[e''_1]}{\Gamma, L_{pc} \vdash e_0[e_1, P]^j \rightsquigarrow {}^{e}/_{\$v_j} : T_P^L}$$

Operations on Type Sets - 3

Inspecting the Type of a Property

$$\upharpoonright (\dot{\tau}, p) = \begin{cases} (\sigma_i, \{\dot{\tau}/\kappa\}\dot{\tau}_p) & \text{if } \dot{\tau} = \mu\kappa.\langle \cdots, p^{\sigma_i} : \dot{\tau}_p, \cdots \rangle^{\sigma} \\ (\sigma_*, \{\dot{\tau}/\kappa\}\dot{\tau}_*) & \text{if } \dot{\tau} = \mu\kappa.\langle \cdots, *^{\sigma_*} : \dot{\tau}_*, \cdots \rangle^{\sigma} \ \wedge \ p \notin dom(\dot{\tau}) \end{cases}$$

Inspecting the Type of a Property Set

$$\upharpoonright_{\uparrow} (\dot{\tau}, P) = \sqcup\{\hat{\sigma} \mid p \in P \wedge \hat{\sigma} = \pi_{\mathtt{lev}}(\upharpoonright (\dot{\tau}, p))\}, \curlyvee\{\dot{\tau}' \mid p \in P \wedge \dot{\tau}' = \pi_{\mathtt{type}}(\upharpoonright (\dot{\tau}, p))\}$$

$$\upharpoonright_{\downarrow} (\dot{\tau}, P) = \sqcap\{\hat{\sigma} \mid p \in P \wedge \hat{\sigma} = \pi_{\mathtt{lev}}(\upharpoonright (\dot{\tau}, p))\}, \curlywedge\{\dot{\tau}' \mid p \in P \wedge \dot{\tau}' = \pi_{\mathtt{type}}(\upharpoonright (\dot{\tau}, p))\}$$

Operations on Type Sets - 4

Inspecting the Type of a Property

$$\Rsh^{?}(\dot{\tau}, P, \$x) = \{(\sigma, \dot{\tau}', (\$x \in \{p\})) \mid p \in P \cap dom(\dot{\tau}) \wedge \Rsh(\dot{\tau}, p) = (\sigma, \dot{\tau}')\}$$

Inspecting the Type of a Property Set

$$\Rsh^{?}(T, P, \$x) = \{(\sigma, \dot{\tau}', \omega \wedge \omega') \mid (\dot{\tau}, \omega) \in T \ \wedge \ (\sigma, \dot{\tau}', \omega') \in \Rsh^{?}(\dot{\tau}, P, \$x)\}$$

Example

Code

$$\mathtt{x}[\mathtt{y}^i] = \mathtt{u}[\mathtt{v}^j] +^k \mathtt{z}$$

Typing Environment

$$\Gamma(\mathtt{x}) = \dot{\tau}_x = \mu\kappa.\langle p_0^L : \mathsf{PRIM}^H, p_1^L : \mathsf{PRIM}^L, *^L : \mathsf{PRIM}^L \rangle^L$$
$$\Gamma(\mathtt{u}) = \dot{\tau}_u = \mu\kappa.\langle q_0^L : \mathsf{PRIM}^L, q_1^L : \mathsf{PRIM}^H, *^L : \mathsf{PRIM}^H \rangle^L$$
$$\Gamma(\mathtt{z}) = \Gamma(\mathtt{y}) = \Gamma(\mathtt{v}) = \mathsf{PRIM}^L$$

Example

Typing Environment

$$\Gamma(\texttt{x}) = \dot{\tau}_x = \mu\kappa.\langle p_0^L : \textsf{PRIM}^H, p_1^L : \textsf{PRIM}^L, *^L : \textsf{PRIM}^L \rangle^L$$
$$\Gamma(\texttt{u}) = \dot{\tau}_u = \mu\kappa.\langle q_0^L : \textsf{PRIM}^L, q_1^L : \textsf{PRIM}^H, *^L : \textsf{PRIM}^H \rangle^L$$
$$\Gamma(\texttt{z}) = \Gamma(\texttt{y}) = \Gamma(\texttt{v}) = \textsf{PRIM}^L$$

Property Types

$$T_{\texttt{x}[\texttt{y}^i]} = \{(\textsf{PRIM}^H, \$v_i \in \{p_0\}), (\textsf{PRIM}^L, \$v_i \in \{p_1\}), (\textsf{PRIM}^L, \neg(\$v_i \in \{p_0, p_1\}))\}$$

$$T_{\texttt{u}[\texttt{v}^j]} = \{(\textsf{PRIM}^L, \$v_j \in \{q_0\}), (\textsf{PRIM}^H, \$v_j \in \{q_1\}), (\textsf{PRIM}^H, \neg(\$v_j \in \{q_0, q_1\}))\}$$

Example

Property Types

$$T_{\mathtt{x}[\mathtt{y}^i]} = \{(\textsf{PRIM}^H, \$v_i \in \{p_0\}), (\textsf{PRIM}^L, \$v_i \in \{p_1\}), (\textsf{PRIM}^L, \neg(\$v_i \in \{p_0, p_1\}))\}$$

$$T_{\mathtt{u}[\mathtt{v}^j]} = \{(\textsf{PRIM}^L, \$v_j \in \{q_0\}), (\textsf{PRIM}^H, \$v_j \in \{q_1\}), (\textsf{PRIM}^H, \neg(\$v_j \in \{q_0, q_1\}))\}$$

Combining Property Sets

$$T_{\mathtt{u}[\mathtt{v}]^j} \oplus_\curlyvee \{(\textsf{PRIM}^L, \textsf{true})\} = T_{\mathtt{u}[\mathtt{v}]^j}$$

$$T_{\mathtt{u}[\mathtt{v}]^j} \oplus_\curlyvee \{(\textsf{PRIM}^H, \textsf{true})\} = \{(\textsf{PRIM}^H, \textsf{true})\}$$

Example

Code

$$\mathtt{x}[\mathtt{y}^i] = \mathtt{u}[\mathtt{v}^j] +^k \mathtt{z}$$

Property Types

$$\text{When}^{?}_{\preceq}(T_{\mathtt{x}[\mathtt{y}^i]}, T_{\mathtt{u}[\mathtt{v}^j]}) = (\$v_i \in \{p_0\})||(\$v_j \in \{q_0\})$$

Instrumented Code

$$\begin{array}{l} \$v_i = \mathtt{y}, \$v_j = \mathtt{v}, \\ (\$v_i == p_0 \,||\, \$v_j == q_0) \; ? \; \big(\mathtt{x}[\$v_i] = \mathtt{u}[\$v_j] + \mathtt{z}\big) : \big(\mathtt{\$diverge}()\big)\big) \end{array}$$

Properties of the Type Systems

Static

- **Soundness:** $\Gamma, \sigma_{pc} \vdash e : \dot{\tau} \;\Rightarrow\; \mathbf{NI}(e, \Gamma)$

Hybrid

- **Soundness:** $\Gamma, L \vdash e \rightsquigarrow {}^{e'}/_{e''} : T \;\Rightarrow\; \mathbf{NI}(e', \Gamma)$
- **Transparancy:** The semantics of the original expression is preserved
- **Optimality:** One cannot gain precision by improving the precision of property set annotations

Future Work

More Expressive Types

- Polimorphic Security Types
- More permissive subtyping relation

Hybrid Mechanism

- Combination of typing with a more expressive logic
- Simplying the generated constraints

Deployment

- Targeting the full language
- Annotating TypeScript with Security levels?