Language Based isolation of Untrusted JavaScript

Ankur Taly

Dept. of Computer Science, Stanford University

Joint work with Sergio Maffeis and John C. Mitchell
1 Introduction

2 Existing subsets of JavaScript: FBJS and ADsafe
   - FBJS\textsubscript{08} : Attacks and Challenges
   - ADsafe\textsubscript{07} : Attacks and Challenges

3 Formal Semantics of JavaScript
   - Structural Operational Semantics
   - Syntax and Key features
   - Semantic rules

4 Achieving the Isolation goal
   - Fundamental Problems
   - Solving the problems : Formal Analysis
   - Fixing FBJS and ADsafe

5 Conclusion and Future Work
What is JavaScript?

- Widely used web programming language.

History:
- Developed by Brendan Eich at Netscape.
- Standardized for Browser Compatibility: ECMAScript 262-edition 3

Some interesting and unusual features:
- First class functions
- Prototype based language
- Powerful modification capabilities: can convert string to code (eval), can redefine object methods!

Very important to fully understand the semantics so as to reason about the security properties of programs written in it.
Many contemporary websites incorporate untrusted JavaScript content:

- Third party advertisements.
- Social Networking sites: User written applications

**Goal**: Allow untrusted code to perform valuable interactions at the same preventing intrusion and malicious damage.

Browser isolation mechanism via Iframes is too restrictive

- Restricts the ad to a delineated section of the page.
- Social network applications need more permissive interaction with the host page

**Statically or Dynamically analyze** untrusted JavaScript code to determine if it is malicious.
Real World Example

**Figure:** Trusted and Untrusted code
Real World Example

Figure: Web Mashup (Valuable Interaction)
Program Analysis Problem

Given an untrusted JavaScript program $P$ and a Heap $H$ (corresponding to the trusted page), design a procedure to either statically or dynamically via run time checks, guarantee that $P$ does not access any security critical portions of the Heap.

- Design static analysis and/or code instrumentation techniques
- Very hard problem to solve for whole of JavaScript as all code that gets executed may not appear textually!
- Example:

  ```javascript
  var m = "toS"; var n = "tring";
  Object.prototype[m + n] = function(){return undefined};
  ```

Approach

Solve the above problem for subsets of JavaScript that are more amenable to static analysis.
Program Analysis Problem

Given an untrusted JavaScript program $P$ and a Heap $H$ (corresponding to the trusted page), design a procedure to either statically or dynamically via runtime checks, guarantee that $P$ does not access any security critical portions of the Heap.

- Design static analysis and/or code instrumentation techniques
- **Very hard problem** to solve for whole of JavaScript as all code that gets executed may not appear textually!
- Example:

  ```javascript
  var m = "toS"; var n = "tring";
  Object.prototype[m + n] = function(){return undefined};
  ```

Approach

Solve the above problem for subsets of JavaScript that are more amenable to static analysis.
Facebook JavaScript (*FBJS*\textsubscript{08})

- **Subset of JavaScript** used for writing Facebook applications.
  - Note:
    - Application code is fetched from the publisher’s (untrusted) server and embedded as a subtree of the page.
    - Not placed in an Iframe.
  - Application code written is **statically checked** to see if it is valid FBJS
  - FBJS code is then re-written and certain **run-time checks** are added.

**Security Goal**

Restrict untrusted code from directly accessing arbitrary elements of the Document Object Model (DOM) and/or the global object.
FBJS: Filtering and Rewriting

Filtering:
- Forbid `eval`, with constructs.
- Disallow explicit access to properties (via the dot notation `-o.p`) `valueOf`, `__parent__`, `constructor`.

Rewriting:
- Add application specific prefix to all top-level identifiers.
  - Example: `o.p` is renamed to `a1234_0.p`
  - Separate effective `namespace` of an application from others.
FBJS: Rewriting contd

- **this** is re-written to `ref(this)`
  - `ref` is a function defined by the host (Facebook) in the global object,
  - `ref(x) = x` if `x \neq window` else `ref(x) = null`
  - Prevents application code from accessing the global object.
- `o[p]` gets rewritten to `o[idx(p)]`.
  - Returns error if `p` is a *black-listed property*, such as, of the form "__x__"

Facebook also provides libraries, accessible within the application namespace, in order to allow applications to safely access certain parts of the global object.
An attack on FBJS

Goal of the Attack
Get a handle to the global object in the application code.

Almost works

```javascript
var getthis = function() { return this; };
```

- `this` gets re-written to `ref(this)` and the code returns `null`.
- Can we re-define `ref` itself?
- `ref` is defined in the **global object** and the application code is disallowed from having handle to global object.
Main Idea:

- Suppose we can get a handle to the current scope object.
- Define a `new ref method` for the current scope object as
  
  \[ \text{ref} = \text{function}(x)\{ \text{return} \ x; \} \]

- Now use the earlier attack code. \text{ref} now behaves as the identity function therefore \text{ref(this)} returns a handle to the window object.
- Once a handle to the window object is obtained all sorts of evil things can be done !.
Demonstrating the vulnerabilities in Firefox
**Attack: Getting the current scope object**

```javascript
try { throw (function() { return this; })(); }
catch (f) { curr_scp = f(); }
```

- ECMA-262 semantics for `try{...} catch(f){...}` says that whenever an exception is thrown:
  - A new object `o` is created with property `f` pointing to the exception object.
  - `o` is placed on top of the scope chain. (`o` does not have the activation object status).
- The "this" of a function not defined in an activation object is the object containing it. In the above code this for `get_scope` resolves to `o`.
- Shadow the original `ref` by re-defining it in `o`.

Ankur Taly

Language Based isolation of Untrusted JavaScript
**Attack: Getting the current scope object**

```javascript
try { throw (function() { return this; })(); }
catch (f) { curr_scp = f(); }
```

- ECMA-262 semantics for `try{...} catch(f){...}` says that whenever an exception is thrown:
  - New object `o` is created with property `f` pointing to the exception object
  - `o` is placed on top of the scope chain. (`o` does not have the activation object status).
  - The "this" of a function not defined in an activation object is the object containing it. In the above code this for `get_scope` resolves to `o`.
  - Shadow the original ref by re-defining it in `o`. 
Attack: Getting the current scope object

```
try { throw (function() {return this;}); }
catch (f) { curr scp = f(); }
```

- ECMA-262 semantics for `try{...} catch(f){...}` says that whenever an exception is thrown:
  - New object `o` is created with property `f` pointing to the exception object
  - `o` is placed on top of the scope chain. (`o` does not have the activation object status).
- The "this" of a function not defined in an activation object is the object containing it. In the above code `this` for `get_scope` resolves to `o`.
- Shadow the original `ref` by re-defining it in `o`.

Ankur Taly
Language Based isolation of Untrusted JavaScript
Recurring subtleties

```javascript
var get_window = function f(x){
    if (x===0) {return this}
    else {f(x - 1)}
};
```

Funny Semantics for recursion.

- ECMA-262 says that whenever a named recursive function `f` is created then the internal scope chain (`f_{scp}`) of the function (environment pointer of the closure) is set to the current lexical scope with a dummy object (`o_f`) placed on top.

  Current Lexical Scope ← [ f : ⟨ code for f ⟩]
Recurring subtleties

```javascript
var get_window = function f(x){
    if (x===0) {return this}
    else {f(x - 1)}
};
```

**Funny Semantics for recursion.**

- ECMA-262 says that whenever a named recursive function $f$ is created then the internal scope chain ($f_{scp}$) of the function (environment pointer of the closure) is set to the current lexical scope with a dummy object ($o_f$) placed on top.

  \[
  \text{Current Lexical Scope} \leftarrow [ f : \langle \text{code for } f \rangle ]
  \]
Recurring subtleties

```javascript
var get_window = function f(x){
  if (x===0) {return this}
  else {f(x - 1)}
};
```

- When the function `f` is called, the current scope chain is replaced with `f_{scp}` and an activation object for `f` is placed on top of it.
- Every recursive call to `f` will resolve to property `f` of the dummy object `o_f` (which is not an activation object).
- Accessing `this` inside `f` will resolve to `o_f`.
- Shadow the original `ref` by re-defining it in `o_f`
ADsafe₀₇ (Douglas Crockford)

- A **safe subset** of JavaScript to be used by **untrusted code not placed in an Iframe**.
- The host page’s server provides the ADSAFE object to the untrusted ad code.
- **All interaction with the trusted code happens only using the methods in the ADSAFE object.**
- Untrusted code can be statically checked to ensure that it only calls methods of the ADSAFE object (*Tool: JSLint*).

**Security Goal**

Restrict untrusted code from directly accessing arbitrary elements of the Document Object Model (DOM) and/or the global object.
ADsafe (First version)

```javascript
var ADSAFE = function () {
    ....
    var reject = function (object, name) {
        return object === window || typeof object !== 'object' ||
        (typeof name !== 'number' &&
        (typeof name !== 'string' || name.charAt(0) == ' '));
    return {
        get: function (object, name) {
            var value;
            if (!reject(object, name) && object.hasOwnProperty(name)) {
                value = object[name];
                if (typeof value !== 'function' && value !== window) {
                    return value;
                }
                error();
            },
            set: function (object, name, value) {
                ....
            }
        }();
    }
}

Usage

To access property \( p \) of object \( o \) : call \( \text{ADSAFE.get}(o, p) \)
Challenges and Issues

- **ADsafe restriction**: "All interaction with the trusted code must happen only using the methods in the ADSafe object.” This may not be possible!
- Consider the following code:

  ```javascript
  // Somewhere in trusted code
  Object.prototype.toNumber = function() { return 1; }
  ...
  // Untrusted code
  var o = {};
  o = o + 3;
  o; // result 4!
  ```

- Reducing `o = o + 3` implicitly calls `Object.prototype.toNumber` on `o`, which is a function defined in the trusted space.
- What if `toNumber` leaks out a pointer to the global object?
Challenges and Issues

- ADsafe restriction: "All interaction with the trusted code must happen only using the methods in the ADSafe object." This may not be possible!

- Consider the following code:

  ```javascript
  // Somewhere in trusted code
  Object.prototype.toNumber = function() { return 1; }
  ...
  // Untrusted code
  var o = {};
  o = o + 3;
  o; // result 4!
  ```

- Reducing `o = o + 3` implicitly calls `Object.prototype.toNumber` on `o`, which is a function defined in the trusted space.

- What if `toNumber` leaks out a pointer to the global object?
Challenges and Issues (contd)

**Summary:**

- We found that a very common JavaScript library - `prototype.js`, provides ways for ADsafe compliant code to access the global object.
- An `eval` method is added to `String.prototype` which allows arbitrary code computed using string manipulations to be executed.

**Example:**

```javascript
var g = "evil_code";
g.evalJSON();
// evalJSON is an eval function added to String.prototype.
```

**Conclusion**

Besides untrusted code, ADsafe07 has to impose restrictions on native functions and objects present in the trusted page's library.
Summary:

- We found that a very common JavaScript library - *prototype.js*, provides ways for ADsafe compliant code to access the global object.
- An `eval` method is added to `String.prototype` which allows arbitrary code computed using string manipulations to be executed.

Example:

```javascript
var g = "evil_code";
g.evalJSON();
// evalJSON is an eval function added to String.prototype.
```

Conclusion

Besides untrusted code, ADsafe07 has to impose restrictions on native functions and objects present in the trusted page’s library.
Summary of our analysis of ADsafe and FBJS

We realize the following three fundamental issues:

- Regardless of the technique adopted, the ultimate goal is to make sure that a piece of untrusted code (that satisfies a certain syntactic criterion), does not access certain global variables.

- There are a number of subtleties related to the expressiveness and complexity of JavaScript.

- It is important to do a fundamental study based on traditional programming language foundations to design provable secure isolation techniques.
1 Introduction

2 Existing subsets of JavaScript: FBJS and ADsafe
   - FBJS\textsubscript{08} : Attacks and Challenges
   - ADsafe\textsubscript{07} : Attacks and Challenges

3 Formal Semantics of JavaScript
   - Structural Operational Semantics
   - Syntax and Key features
   - Semantic rules

4 Achieving the Isolation goal
   - Fundamental Problems
   - Solving the problems : Formal Analysis
   - Fixing FBJS and ADsafe

5 Conclusion and Future Work
Informal Semantics

- **ECMA262-3** specification manual - currently in its third edition.
- Sufficient for 'understanding' the language but insufficient for rigorously proving properties about the language.
- Prove or Disprove: For all terms \( t \), the execution of \( t \) only depends on the values of the variables appearing in \( t \).
  - Example: \( Meaning[x = x + 10] \) depends only on value of \( x \)?
  - in C? Yes
  - in JavaScript?
A corner case

```javascript
var y = "a";
var x = {toString: function(){ return y;}}

x = x + 10;
jsx> "a10"
```

- Implicit type conversion of an object to a string in JavaScript involves calling the `toString` function.
- Informal semantics fail to bring out such corner cases.
Formal Semantics

- Specify meaning in a **Mathematically rigorous** way.
- Provides a framework for proving properties of the kind mentioned on the previous slide.

**Our Goal**

- Convert Informal semantics (ECMA262-3) into a Formal semantics. (Done! This talk)
- Analyze existing **safe subsets** of JavaScript and formally prove the security properties that they entail. (Ongoing work)

- The very act of formalization revealed **subtle aspects** of the language.
Basic JavaScript Syntax

Syntax

According to ECMA 2.62:

Expressions (e)  ::  this | x | e OP e | e(e) | new e(e) | ...

Statement (s)  ::  "s*" | if (e) s else s |

Programs (P)  ::  s P | fd P

Function Decl (fd)  ::  function x (x){ P }

Observation

Observe that according to the spec, declaring a function inside an 'if block' is a syntax error! However, this is allowed in all browsers.
Introduction

Existing subsets of JavaScript: FBJS and ADsafe

Formal Semantics of JavaScript

Achieving the Isolation goal

Conclusion and Future Work

Basic JavaScript Syntax

Syntax

According to ECMA 2.62:

- **Expressions** (e) :: this | x | e OP e | e(e) | new e(e) | ...
- **Statement** (s) :: "s*" | if (e) s else s | while (e) s | with (e) s | ...
- **Programs** (P) :: s P | fd P
- **Function Decl** (fd) :: function x (x){ P }

Observation

Observe that according to the spec, declaring a function inside an 'if block' is a **syntax error**! However this allowed in all browsers.
Structural Operational Semantics (Gordon Plotkin)

- Meaning of a program ⇔ sequence of actions that are taken during its execution.
- Specify sequence of actions as transitions of an Abstract State machine.
- States corresponds to
  - Term being evaluated
  - Abstract description of memory and other data structures involved in computation.
- A state transition denotes a partial evaluation of the term.
- Specify the transitions in a syntax oriented manner using the inductive nature of the abstract syntax.
JavaScript: Key Features

- Everything (including functions) is either an object or a primitive value.
- Prototype based inheritance mechanism.
- **Activation records are normal JavaScript objects** and the variable declarations are properties of this object.
- All computation happens inside a **global object** which is also the initial activation object.
- Instead of a stack of activation records, there a chain of activation records, which is called the **scope chain**.
- Arbitrary objects can be placed over the scope chain - with(e) s construct.
Example 2

var f = function() {
if (true) {
  function g() {
    return 1;
  };

else {
  function g() {
    return 2;
  };
}

var g = function() {
  return 3;
}

return g();

function g() {
  return 4;
}

var result = f();

What is the final value of result?
JavaScript : Subtle Features

Example 2

```javascript
var f = function() {
    if (true) {
        function g() {
            return 1;
        };
    } else {
        function g() {
            return 2;
        };
    }
    var g = function() {
        return 3;
    }
    return g();
    function g() {
        return 4;
    }
};

var result = f();
```

What is the final value of `result`?

- `result = 2` (according to ECMA262-3)
- Function body is parsed to process all variable declarations before the function call is executed!
- Different implementations chose different declarations: Mozilla Spidermonkey: 4, Safari: 1!
Formal Semantics : Program state

- All objects are passed by reference ⇒ The store must have information about Heap locations.
- Variables have different values in different scopes ⇒ State must include info about current scope.

State

Program state is represented as a triple \( \langle H, l, t \rangle \).

- \( H \) : Denotes the Heap, mapping from the set of locations \( \mathbb{L} \) to objects.
- \( l \) : Location of the current scope object (or current activation record).
- \( t \) : Term being evaluated.
Semantic Rules

- Three semantic functions $e \rightarrow$, $s \rightarrow$, $P \rightarrow$ for expressions, statements and programs.

- Small step transitions: A semantic function transforms one state to another if certain conditions (premise) are true.

- General form: \[
\frac{\langle \text{Premise} \rangle}{S \xrightarrow{t} S'}
\]

- Atomic Transitions: Rules which do have another transition in their premise (Transition axioms).

- Context rules: Rules to apply atomic transitions in presence of certain specific contexts.
Heap and Heap Reachability Graph

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_0^A$</td>
<td>@scope: #global</td>
</tr>
<tr>
<td>#global</td>
<td>“a”: $l_1^A$</td>
</tr>
<tr>
<td>$l_1^A$</td>
<td>“n”: $l_2^A$ “p”: $l_3^A$</td>
</tr>
<tr>
<td>$l_2^A$</td>
<td>“value”: 10</td>
</tr>
<tr>
<td>$l_3^A$</td>
<td>“value”: 20</td>
</tr>
<tr>
<td>$l_4^A$</td>
<td>“@scope”: $l_0^A$</td>
</tr>
</tbody>
</table>

Heap Reachability Graph: Heap addresses are the nodes. An edge from $l_i$ to $l_j$, if the object at address $l_i$ has property $p$ pointing to $l_j$.

Figure: Heap and its reachability graph
Each Heap object $o$ is a record $\{p_1 : ov_1, \ldots, p_n : ov_n\}$ where $p_i$ are property names and $ov_i$ could be a primitive value or another heap addresses. Meta-functions:

- $Dot(H, l, p) = l_1$ : Access the property $p$ of object at heap location $l$.
- $Put(H, l, p, l_v) = H'$ : Update the property $p$ of object at $H(l)$ and return the new Heap.
- $H', l = alloc(H, o)$ : Allocate object $o$ to a new heap location $l$. 
Identifier Resolution

Some Notation :

- o "hasProperty" p ⇔ "p" is a property of object "o" or one of the ancestral prototypes of "o".
- o "hasOwnProperty" p ⇔ "p" is a property of object "o" itself.

- A JavaScript reference type is pair denoted as \( I \times p \) where \( I \) is heap address, also called the base type of the reference, and \( p \) is a property name.

**Procedure**

- Traverse down the scope chain.
- For each object \( o \) (heap address \( l_o \)),
  - If \( o \)"hasproperty" \( x \), return reference \( l_o \times x \)
  - Else move to the next object.
Every scope chain has the **global object at its base**.

Every prototype chain has **Object.prototype at the top**, which is a native object containing predefined functions such as `toString`, `hasOwnProperty` etc.

**Figure**: Scope and Prototype lookup
Scope lookup: Rules

ECMA 2.62:

1. Get the next object (l) in the scope chain. If there isn’t one, goto 4.
2. If l "HasProperty" x, return a reference type l*"x".
3. Else, goto 1
4. Return null*x.

\[
\text{Scope}(H, l, "x") = l_n
\]

\[
\langle H, l, x \rangle \xrightarrow{e} \langle H, l, l_n \ast "x" \rangle
\]

\[
\text{HasProperty}(H, l, m)
\]

\[
\text{Scope}(H, l, m) = l
\]

\[
\neg (\text{HasProperty}(H, l, m))
\]

\[
H(l).\text{@Scope} = l_n
\]

\[
\text{Scope}(H, l, m) = \text{Scope}(H, l_n, m)
\]

\[
\text{Scope}(H, \text{null}, m) = \text{null}
\]
Prototype lookup: Rules

ECMA 2.62:

1. If base type is null, throw a ReferenceError exception.
2. Else, Call the Get method, passing prop name(x) and base type l as arguments.
3. Return result(2).

\[ H_2, l_{\text{excp}} = \text{alloc}(H, o) \]
\[ o = \text{newNativeErr}('', \#\text{RefErrProt}) \]
\[ \langle H, l, (\text{null} \ast m) \rangle \xrightarrow{e} \langle H_2, l, \langle l_{\text{excp}} \rangle \rangle \]

\[ \text{Get}(H, l, m) = \text{va} \]
\[ \langle H, l, l_{\text{in}} \ast m \rangle \xrightarrow{e} \langle H, l, \text{va} \rangle \]

\[ \text{HasOwnProperty}(H, l, m) \]
\[ \text{Dot}(H, l, m) = \text{va} \]
\[ \text{Get}(H, l, m) = \text{va} \]

\[ \neg (\text{HasOwnProperty}(H, l, m)) \]
\[ H(l).@\text{prototype} = lp \]
\[ \text{Get}(H, l, m) = \text{Get}(H, lp, m) \]
Exceptions

- When an intermediate step gives an exception, stop further evaluation and throw the exception to the top level.
- Example:

\[
\langle H, l, a_0 \rangle \rightarrow \langle H, l, \langle l_{\text{excp}} \rangle \rangle \\
\langle H, l, a_0 + a_1 \rangle \rightarrow \langle H, l, \langle l_{\text{excp}} \rangle + a_1 \rangle
\]

Stop evaluation of $a_2$.
- Use context based reduction rules (Felleisen)

Context Rule for Exceptions

\[
\langle H, l, eC[\langle l_{\text{excp}} \rangle] \rangle \rightarrow \langle H, l, \langle l_{\text{excp}} \rangle \rangle \\
\text{where } eC ::= _ | eC \text{ OP } e | \text{ va } \text{ OP } eC | eC[e] | \ldots
\]
Exceptions

• When an intermediate step gives an exception, stop further evaluation and throw the exception to the top level.

• Example:

\[
\langle H, l, a_0 \rangle \rightarrow \langle H, l, \langle \text{lecp} \rangle \rangle
\]

\[
\langle H, l, a_0 + a_1 \rangle \rightarrow \langle H, l, \langle \text{lecp} \rangle + a_1 \rangle
\]

Stop evaluation of \( a_2 \).

• Use context based reduction rules (Felleisen)

Context Rule for Exceptions

\[
\langle H, l, eC[\langle \text{lecp} \rangle] \rangle \rightarrow \langle H, l, \langle \text{lecp} \rangle \rangle
\]

where \( eC ::= _- \mid eC \text{ OP } e \mid va \text{ OP } eC \mid eC[e] \mid \ldots \)
Summary

- We developed an operational semantics for the entire ECMA 2.62 language.
- Complete set of rules (in ASCII) span 70 pages.
- Semantics does not cover features beyond ECMA 2.62, like setters/getters etc, which are present in various browsers.
- We do model interaction with the Document Object Model (DOM) of web browsers.
- The entire exercise also led to the discovery of several inconsistencies in the various browsers.
1 Introduction

2 Existing subsets of JavaScript: FBJS and ADsafe
   - FBJS$_{08}$: Attacks and Challenges
   - ADsafe$_{07}$: Attacks and Challenges

3 Formal Semantics of JavaScript
   - Structural Operational Semantics
   - Syntax and Key features
   - Semantic rules

4 Achieving the Isolation goal
   - Fundamental Problems
   - Solving the problems: Formal Analysis
   - Fixing FBJS and ADsafe

5 Conclusion and Future Work
Back to the problem

Central Problem

Ensure that a piece of untrusted code written in a safe subset does not access certain global variables.

Two Approaches:

- **Filtering**: Enforce syntactic restrictions and guarantee safety at static time.
  - Robust and Efficient
  - Semantics of user code is unaltered.

- **Rewriting**: Appropriately rewrite certain parts of the user code (recall \texttt{idx} and \texttt{ref}) to ensure that certain global variables are not accessed.
  - Run time overhead
  - Ensuring that user semantics is preserved is non-trivial
  - Formal analysis is more involved.

This work: **Filtering**.
Two Problems

Problem 1

Design a meaningful sublanguage $Jt$ so that given a program $P \in Jt$, we can statically determine a finite set $\text{Access}(P)$ which is the list of all property names that can be potentially accessed.

Once this problem is solved

- Isolation of library code from untrusted code becomes easy.
- Create a Blacklist $B$ of all security critical properties of the global object.
- Filter $P$ if $\text{Access}(P) \cap B \neq \emptyset$.

What about isolation of code from other untrusted sources

- For two untrusted program $P_1$ and $P_2$ from different sources, we need to enforce $\text{Access}(P_1) \cap \text{Access}(P_2) = \emptyset$.
- Too strong!
Two Problems

Problem 1

Design a meaningful sublanguage $Jt$ so that given a program $P \in Jt$, we can statically determine a finite set $\text{Access}(P)$ which is the list of all property names that can be potentially accessed.

Once this problem is solved

- **Isolation of library code from untrusted code becomes easy.**
- Create a Blacklist $\mathcal{B}$ of all security critical properties of the global object.
- Filter $P$ if $\text{Access}(P) \cap \mathcal{B} \neq \emptyset$.

What about isolation of code from other untrusted sources

- For two untrusted program $P_1$ and $P_2$ from different sources, we need to enforce $\text{Access}(P_1) \cap \text{Access}(P_2) = \emptyset$.
- **Too strong !**
Two Problems

Problem 1

Design a meaningful sublanguage $Jt$ so that given a program $P \in Jt$, we can statically determine a finite set $\text{Access}(P)$ which is the list of all property names that can be potentially accessed.

Once this problem is solved

- Isolation of library code from untrusted code becomes easy.
- Create a Blacklist $B$ of all security critical properties of the global object.
- Filter $P$ if $\text{Access}(P) \cap B \neq \emptyset$.

What about isolation of code from other untrusted sources

- For two untrusted program $P_1$ and $P_2$ from different sources, we need to enforce $\text{Access}(P_1) \cap \text{Access}(P_2) = \emptyset$.
- Too strong!
Two Problems

One Possible Solution: Renaming identifiers in order to separate the namespace of untrusted code.

- Example: rename \( x = x + 5 \) to \( a123\_x = a123\_x + 5 \).
- Don’t rename property names, \( o.p \) is renamed to \( a123\_o.p \) and not \( a123\_o.a123\_p \).

But does this preserve the semantics?

- Not for \( J_t \).
- Issue: Variables are essentially properties of the current scope object (activation object).
  - \( \text{var } x = 42; \ this.x \text{ returns } 42 \) when executed in the global scope.
  - \( \text{var } a123\_x = 42; \ this.x \text{ returns } "\text{Reference error } x \text{ not defined}" \).
Two Problems

**One Possible Solution**: Renaming identifiers in order to separate the namespace of untrusted code.

- Example: rename `x = x + 5` to `a123.x = a123.x + 5`.
- Don’t rename property names, `o.p` is renamed to `a123.o.p` and not `a123.o.a123.p`.

But does this preserve the semantics?

- Not for `Jt`.
- **Issue**: Variables are essentially properties of the current scope object (activation object).
  - `var x = 42; this.x` returns 42 when executed in the global scope.
  - `var a123.x = 42; this.x` returns "Reference error x not defined".
Two problems

Problem 2

Define a meaningful sublanguage $Js$ so that no program $P \in Js$ can return a pointer to a scope object.

Once this problem is solved:

- Isolation from other untrusted code can be achieved.
- Appropriately rename all user defined identifier names appearing in the code.
- It turns out that our solution $Js$ satisfies $Js \subseteq Jt$. 
Solving Problem 1 - Isolation of property names

Textual Property

Given a program $P$, all property names that get accessed must appear textually in the code.

Issues:

- **Intractable in general**: Code that gets executed may not appear textually.
  
  ```javascript
  var o = { prop: 42 }; m = "pr"; n = "op"; o[m+n]
  ```

- Get rid of `eval`, `Function`, `o[..]` etc which convert string to code.

- Recall: **Implicit property accesses**
  
  ```javascript
  var o = new Object();
  var x = 4;
  x = o + 3;
  ```

  `toNumber` property of $o$ is accessed implicitly!
Solving Problem 1- Isolation of Property names

- Calling `Array.prototype` methods involve implicit access to numeric properties ("1", "2", ...).
- From the semantics, we get the following list of property names that can get accessed implicitly. \( \mathcal{P}_{nat} = \{0,1,2,...\} \cup \{\text{toString, toNumber, valueOf, length, prototype, constructor, message, arguments, Object, Array}\} \)
- Assume accessing properties in \( \mathcal{P}_{nat} \) is safe.

Textual Property (revised)
Define a meaningful sublanguage \( Jt \) such that for any program \( P \in Jt \), if execution of \( P \) accesses property \( p \) of some object, then either \( p \in \text{Prop}_{nat} \) or \( p \) appears textually in \( P \).
Solving Problem 1- Isolation of Property names

- Calling `Array.prototype` methods involve implicit access to numeric properties ("1", "2", ...).
- From the semantics, we get the following list of property names that can get accessed implicitly. \( \mathcal{P}_{nat} = \{0,1,2,...\} \cup \{\text{toString, toNumber, valueOf, length, prototype, constructor, message, arguments, Object, Array}\} \)

- Assume accessing properties in \( \mathcal{P}_{nat} \) is safe.

**Textual Property (revised)**

Define a meaningful sublanguage \( Jt \) such that for any program \( P \in Jt \), if execution of \( P \) accesses property \( p \) of some object, then either \( p \in \text{Prop}_{nat} \) or \( p \) appears textually in \( P \).
Sublanguage $J_t$

Designing the subset:

- **Fundamental issue**: Strings ($m$), Property Names ($pn$) and Identifiers ($x$) are implicitly converted to each other
- Separate them out and track the inter conversion in the semantics.
- Terms whose reduction trace involves conversion from
  - Strings $\rightarrow$ Property names (like $e[e]$)
  - Strings $\rightarrow$ Code (like `eval`)

are evil. Get rid of them!

**Defining $J_t$**

$J_t$ is defined as ECMA-262 MINUS: all terms containing the identifiers `eval`, `Function`, `hasOwnProperty`, `propertyIsEnumerable` and `constructor`; expressions $e[e]$, $e$ in $e$; the statement for ($e$ in $e$) s;
Formal Analysis

Notation:

- $\tau(S)$: Reduction trace of state $S = S_1 \rightarrow, \ldots, \rightarrow S_n, \rightarrow \ldots$.
- $\mathcal{A}(S)$: Set of property names that get accessed during a single step state transition. Extended naturally to traces.
- $\mathcal{N}_i^T(S)$: Set of all identifiers appearing in state $S$.

Textual Property $Pt$ : Formally

Given a state $S$, $Pt(S)$ holds iff

$$\mathcal{A}(\tau(S)) \subseteq \text{Id2Prop}(\mathcal{N}_i^T(S)) \cup P_{nat}.$$ 

Theorem 1 (Isolating property names)

For all well-formed states $S_0$ in $\text{Initial}(Jt)$, $Pt(S_0)$ holds.
Proof: Approach

**Step 1:** We define a stronger textual property $P_t^{\text{strong}}$ and show that for all initial states $S_0$ in JavaScript, $P_t^{\text{strong}}(S_0) \Rightarrow P_t(S_0)$

- Informally, this step shows that not invoking certain bad reduction rules is sufficient for showing that the textual property holds.

**Step 2:** For all initial states $S_0$ in the subset $Jt$, $P_t^{\text{strong}}(S_0)$ holds.

- Informally, this step shows that our definition of $Jt$ is such that none of these bad reduction rules are invoked.
Proof: Step 1

Notation:

- Given a trace $\tau(S) = S_1 \rightarrow \ldots \rightarrow S_n \rightarrow \ldots$, $\mathcal{R}(\tau(S))$ is the set of all transition axioms used to derive $S_i \rightarrow S_{i+1}$ (for all $i$).
- Let $\mathcal{R}^{bad}$ be the set of all reduction rules which involve one of the following conversions:
  - strings to property names (e[e] constructs).
  - strings to identifiers (Function constructor).
  - strings to programs (eval construct).

Strong textual property ($Pt^{strong}$)

For a given state well-formed initial state $S_0$ we define $Pt^{strong}(S_0)$ as true iff $\mathcal{R}(\tau(S_0)) \cap \mathcal{R}^{bad} = \emptyset$.

Lemma 1

For all initial states $S_0$ in JavaScript, $Pt^{strong}(S_0) \Rightarrow Pt(S_0)$.
Proof: Step 1

Notation:
- Given a trace $\tau(S) = S_1 \rightarrow \ldots \rightarrow S_n \rightarrow \ldots$, $R(\tau(S))$ is the set of all transition axioms used to derive $S_i \rightarrow S_{i+1}$ (for all $i$).
- Let $R^{bad}$ be the set of all reduction rules which involve one of the following conversions:
  - strings to property names ($e[e]$ constructs).
  - strings to identifiers (Function constructor).
  - strings to programs (eval construct).

Strong textual property ($Pt^{strong}$)

For a given state well-formed initial state $S_0$ we define $Pt^{strong}(S_0)$ as true iff $R(\tau(S_0)) \cap R^{bad} = \emptyset$.

Lemma 1

For all initial states $S_0$ in JavaScript, $Pt^{strong}(S_0) \Rightarrow Pt(S_0)$
Proof : Step 2

- We show that for all initial states $S_0$ in the subset $Jt$, $Pt^{strong}(S_0)$ holds.

**Idea** : Define a **Goodness property** (Inductive Invariant) on states such that :
- A state being good implies that no reduction rule from the set $R^{bad}$ applies.
- Goodness is preserved under reduction.
Proof : Step 2

**Term goodness**

We say that a term $t$ is *good*, denoted by $\text{Good}_{Jt}(t)$ iff it has the following properties:

- **Structure of $t$ does not contain any of** `eval`, `Function`, `hasOwnProperty`, `propertyIsEnumerable` **and constructor as property names or identifiers**.
- **Structure of $t$ does not contain any sub terms with any contexts of the form** `e for in e`, `p for in e`, `cEval()`, `FunParse()` **or** `[]` **contexts and any constructs of the form** `e in e`, `for (e in e)` **s and** `e[e]`.
- **Structure of $t$ does not contain any of the heap addresses** $l_{\text{Function}}$, $l_{\text{eval}}$, $l_{\text{hOP}}$, $l_{\text{pIE}}$. 
Heap goodness

We say that a heap is *good*, denoted by $Good_{\text{t}}(H)$, iff $H$ has the following properties

\[
\forall l, p : H(l).p = l_{\text{Function}} \Rightarrow p = \text{constructor}
\]

\[
\forall l, p : H(l).p = l_{\text{eval}} \Rightarrow p = \text{eval}
\]

\[
\forall l, p : H(l).p = l_{\text{hOP}} \Rightarrow p = \text{hasOwnProperty}
\]

\[
\forall l, p : H(l).p = l_{\text{pIE}} \Rightarrow p = \text{propertyIsEnumerable}
\]
Proof: Step 2

Lemma 2 (Goodness implies Safety)

For all well-formed states $S$ in the subset $Jt^*$ such that $Good_{Jt}(S)$ is true, no reduction rule from $R^{bad}$ applies to $S$.

Lemma 3 (Goodness preserved under reduction)

For all well-formed states $S_1$ and $S_2$ in the subset $Jt^*$, $S_1 \rightarrow S_2 \land Good_{Jt}(S_1) \Rightarrow Good_{Jt}(S_2)$

Lemma 4 (Goodness holds for all initial states)

For all well-formed states $S_0$ in $Initial(Jt)$, $Good_{Jt}(S_0)$ is true.
Isolating scope object

- For initial empty heap state, global object is only accessible via `@scope` and `@this` properties.
- Other than dereferencing `@this` there is no way of returning the current scope object at the top level.
- `Object.prototype.valueOf`, `Array.prototype.sort/concat/reverse` can potentially deference the `@this` property.

### Defining Js

The subset $Js$ is defined as $Jt$, MINUS: all terms containing the expression `this`; all terms containing the identifiers `valueOf`, `sort`, `concat` and `reverse`;

$Js \subset Jt$ is sufficient for imposing the restriction that the identifiers `valueOf`, `sort`, `concat` and `reverse` do not appear in the program during its reduction.
Isolating scope object

- For initial empty heap state, global object is only accessible via @scope and @this properties.
- Other than dereferencing @this there is no way of returning the current scope object at the top level.
- `Object.prototype.valueOf`, `Array.prototype.sort/concat/reverse` can potentially deference the @this property.

Defining Js

The subset Js is defined as Jt, MINUS: all terms containing the expression this; all terms containing the identifiers `valueOf`, `sort`, `concat` and `reverse`;

Js ⊂ Jt is sufficient for imposing the restriction that the identifiers `valueOf`, `sort`, `concat` and `reverse` do not appear in the program during its reduction.
Formal Analysis

Notation:
- \( \text{Final}(S) \): Final term in \( \tau(S) \) (if it exists)
- \( \mathcal{V}(S') \): Value part of the state (if it exists).

Property \( Ps \)

Given a state \( S \), let \( S' = \text{Final}(S) \). \( Ps(S) \) holds iff value part of \( S' \) is not the address of a scope object.

Theorem 2 (Isolation of scope object)

For all well-formed states \( S_0 \) in \( \text{Initial}(Js) \), \( Ps(S_0) \) holds.
Proof: Approach

Similar to proof of Theorem 1

- Define a **Goodness property** on states such that
  - Goodness is preserved under reduction.
  - State goodness implies that the value part is never the address of a scope object.

**Lemma 5 (Goodness preserved under reduction)**

For all well-formed states $S_1$ and $S_2$ in the subset $Js^*$,

$S_1 \rightarrow S_2 \land \text{Good}_{Js}(S_1) \Rightarrow \text{Good}_{Js}(S_2)$

**Lemma 6 (Goodness holds for initial state)**

For all well-formed states $S_0$ in $\text{Initial}(Js)$, $\text{Good}_{Js}(S_0)$ is true.
Additional Property: Identifier Renaming

- No program in subset `Js` can get a handle to a scope object.
- Therefore **variable names can never be accessed (explicitly) as properties**.
- As a result, variable names can be renamed without modifying the semantics.
- **Catch**: Don’t rename variable with same name as native properties.
  - `toString()` evaluates to `"[object Window]"`
  - `a123.toString()` evaluates to `Reference error`.  
  - Problem: Native `toString` method not renamed on the heap.

**Theorem 3 (Closure under renaming)**

For all well-formed states $S_0$ in `Initial(Js)`, if $\alpha$ is a safe renaming function with respect to $S_0$, then $\alpha(\tau(S_0))$ equals $\tau(\alpha(S_0))$. 

A solution for mitigating FBJSAttack

- **Main issue**: `ref` can be re-defined.
- Define `$FBJS.ref` in a different name-space.
- Make sure that application code does not access any property names involving "$".
- Within hours of our discovery, *Facebook* addressed the problems we discovered and adopted the above fix !.
A purely syntactic solution for FBJS

We propose $Js$ as a purely syntactic solution for FBJS.

- Straightforward blacklisting of security critical properties will isolate untrusted code from library code.
- Supports variable renaming so untrusted code form different applications can be isolated.
- **Downside**: Might be too conservative.
A solution for ADsafe

Shortly after we notified Yahoo!, the ADsafe document was amended with the additional constraint:

“None of the prototypes of the built-in types may be augmented with methods that can breach ADsafe’s containment”

Our proposal

- Analyze the library of the host page and extract a blacklist $P_{noRW}$ such that all illicit access to security critical properties happens via a property in $P_{noRW}$ (Open problem !)
- More corrupt the library code is, larger would be $P_{noRW}$.
- Use subset $Js + Filter$ for $P_{noRW}$. 
Summary

- FBJS\textsubscript{08} and ADsafe\textsubscript{07} have the same security goals.
- FBJS has an advantage that it is the first player so it gets to set the host page library.
- ADsafe\textsubscript{07} must work with an arbitrary host library which might be compromised.
- Therefore ADsafe has to be much more restrictive than FBJS.
Conclusions and Future Work

- Write the semantics in **machine readable** format so that the proofs can be automated.
- Extend the above results to apply to JavaScript supported by various browsers which include features beyond the ECMA-262 spec, such as getter, setters etc.
- Design a static type inferencing system for a suitable sub-language so that besides property names we can also reason about the **objects involved in Heap accesses**.
- Develop a feasible procedure to analyze the library code in order to completely solve the ADsafe problem.
Thank You!