2011 IEEE Symposium on Security and Privacy

Automated Analysis of Security-Critical JavaScript APIs

Ankur Taly
Stanford University
ataly@stanford.edu

Úlfar Erlingsson
Google Inc.
ulfar@google.com

John C. Mitchell
Stanford University
mitchell@stanford.edu

Mark S. Miller
Google Inc.
erights@google.com

Jasvir Nagra
Google Inc.
jasvir@google.com

***Abstract*—JavaScript is widely used to provide client-side functionality in Web applications. To provide services ranging from maps to advertisements, Web applications may incorporate untrusted JavaScript code from third parties. The trusted portion of each application may then expose an API to untrusted code, interposing a reference monitor that mediates access to security-critical resources. However, a JavaScript reference monitor can only be effective if it cannot be circumvented through programming tricks or programming language idiosyncrasies. In order to verify complete mediation of critical resources for applications of interest, we define the semantics of a restricted version of JavaScript devised by the ECMA Standards committee for isolation purposes, and develop and test an automated tool that can soundly establish that a given API cannot be circumvented or subverted. Our tool reveals a previously-undiscovered vulnerability in the widely-examined Yahoo! ADsafe filter and verifies confinement of the repaired filter and other examples from the Object-Capability literature.**

***Keywords*-Language-Based Security, Points-to Analysis, APIs, Javascript**

I. Introduction

JavaScript is widely used to provide client-side functionality in Web applications. Many contemporary websites incorporate untrusted third-party JavaScript code into their pages in order to provide advertisements, Google Maps, so-called gadgets, and applications on social networking websites. Since JavaScript code has the ability to manipulate the page Document Object Model (DOM), steal cookies, and navigate the page, untrusted third-party JavaScript code may pose a significant security threat to the hosting page.

While third-party code may be isolated by placing it in an iFrame, this reduces performance and restricts interaction between the hosting page and third-party code. Instead, Facebook and other sites rely on language-based techniques [36] to embed untrusted applications directly into the hosting page.

A widely used approach combines a language-based sandbox to restrict the power of untrusted JavaScript with trusted code that exports an API to untrusted code. In the *API+Sandbox* approach, used in Facebook FBJS [36], Yahoo! ADsafe [9], and Google Caja [6], the trusted code must encapsulate all security-critical resources behind an API that provides JavaScript methods to safely access these resources. While there has been significant progress [23–25] toward provably-safe sandboxes for restricting access to the global object and other critical objects, very little research has been devoted to rigorously analyzing API confinement. In this paper, we therefore study and provide precise semantics for a subset of JavaScript that supports confinement, present an automated tool that provably verifies confinement, and use this tool to analyze code designed to provide confinement.

We consider a variant of a recently-standardized version of JavaScript that supports static scoping and hiding of nested local variables. Using this language, our static analysis method examines *trusted code* used in a hosting page, such as security-focused wrapping libraries, and determines whether it is secure against arbitrary untrusted code in the same language. Since trusted code enforcing a reference monitor is written by security-conscious programmers, we believe the authors of trusted code may be willing to structure their code to improve the precision of the analysis. Under these conditions, our automated method is sufficient to verify that no interleaved sequence of API method calls returns a direct reference to a security-critical object.

Given an implementation of an API reference monitor and a set of security-critical objects, our automated tool `ENCAP` soundly verifies API confinement. We used this tool to analyze the Yahoo! ADsafe library [9] under the threat model defined in this paper and found a previously undetected security oversight that could be exploited on the current web to leak access to the `document` object (and hence the entire DOM tree). This demonstrates the value of our analysis, as ADsafe is a mature security filter that has been subjected to several years of scrutiny and even automated analysis [18]. After repairing the vulnerability, our tool is sufficient to prove confinement of the resulting library under our threat model.

A. API+Sandbox examples

We illustrate the API+Sandbox approach using a simple example: a hosting page intends to provide a *write only* log facility to untrusted code. It enforces this intent by creating an array to log data and an API object which has a single method `push` that only allows data to be pushed on to the array. The API object is then provided to untrusted code by placing it in a global variable `api`.

```
var priv = criticalLogArray;
var api = {push: function(x){priv.push(x)}}
```

1081-6011/11 $26.00 © 2011 IEEE
DOI 10.1109/SP.2011.39

Untrusted code is restricted so that the only global variable accessible to it is api. A necessary requirement in establishing correctness of this mechanism is to verify that the API object does not leak a direct reference to criticalLogArray as that would allow reading data from the array.

While the example above may suggest that the API confinement problem is easily solved, the addition of the following store method to the API may suggest otherwise:

```
api.store = function(i,x){priv[i] = x}
```

While a cursory reading shows that neither API method returns a reference to the array, the API fails to confine the array. A client may gain direct access to the criticalLogArray by calling methods of the API and mutating external state, as in the following code:

```
var result;
api.store('push',function(){result = this[0]});
api.push();
```

The exploit makes unanticipated use of the store method by supplying "push" as the first argument instead of a numeral. Our automated analysis detects this problem by effectively considering all possible invocations of all the API methods.

The foundations of the API+Sandbox approach lie in the *object-capability* theory of securing systems (see [20, 29]). In the context of capabilities, the methods of the API are *capabilities* supplied to untrusted code and the sandbox is the *loader* that loads untrusted code only with a given set of capabilities [29]. If API methods are viewed as capabilities, then the *API Confinement Problem* is also known as the *Overt Confinement Problem for Capabilities* [19].

B. Confinement-friendly JavaScript

One reason why prior work has not focussed on verifying correctness of APIs is because present JavaScript, based on the 3^{rd} edition of ECMA-262 standard, is not amenable to static analysis, for reasons discussed in section 2. Recognizing these difficulties, the ECMA Standards Committee (TC39) developed a *strict mode* (ES5S) in the 5^{th} edition of the ECMAScript Standard (ES5) [10], that supports static lexical scoping and closure-based encapsulation. ES5S, however, has two remaining limitations for confinement and static analysis: (1) ambient access to built-in objects may be used to subvert some of the checks in API implementations, and (2) eval allows dynamic code execution.

In this paper, we propose a variant SES$_{light}$ of ES5S that supports static analysis and confinement by eliminating the two problems above. As discussed in section 2, SES$_{light}$ is comparable to and more expressive than previous JavaScript sandboxing sublanguages. In SES$_{light}$, malicious use of built-in objects is restricted by making necessary objects immutable. For dynamic code execution, the language only supports a restrictive form of eval, which we call *variable-restricted eval*, that is amenable to static analysis. While a more permissive sublanguage *Secure EcmaScript* (SES) is currently under proposal by the ECMA committee (TC 39), the two languages are relatively close. The main difference between SES and SES$_{light}$ is that SES supports getters/setters and SES$_{light}$ does not because they are not amenable to the analysis methods we considered practical in developing our approach. Since no current browser implements the SES$_{light}$ semantics, we describe a way to enforce the SES$_{light}$ semantics in an ES5S environment, using an initialization script that must be run in the beginning and a static verifier that must be applied to all code that runs subsequently. While we have implemented this method, our formal analysis is based on the independent semantics of SES$_{light}$

C. Static analysis method

The main technique used in our verification procedure is a conventional context-insensitive and flow-insensitive points-to analysis. We analyze the API implementation and generate a conservative Datalog model of all API methods. We encode an attacker as a set of Datalog rules and facts, whose consequence set is an abstraction of the set of all possible invocations of all the API methods. Our attacker encoding is similar to the encoding of the conventional Dolev-Yao network attacker, used in network protocol analysis. We prove the soundness of our procedure by showing that the Datalog models for the API and the attacker are sound abstractions of the semantics of the API and the set of all possible sandboxed untrusted code respectively. The threat model is based on arbitrary untrusted SES$_{light}$ code run with respect to the SES$_{light}$ semantics after sandboxing.

D. Contributions and Organization

In summary, the main contributions of this paper are:

- The syntax and semantics of the language SES$_{light}$, which supports a safe sandbox and is amenable to static analysis,
- A Datalog-based procedure for deciding confinement properties of SES$_{light}$ APIs,
- A proof of semantic soundness of the procedure under the SES$_{light}$ threat model,
- An implementation of the procedure in the form of an automated tool ENCAP, and
- Applications of the tool to demonstrate an attack on Yahoo! Adsafe, confinement properties of repaired Adsafe, and confinement properties of standard examples from the object-capability literature.

The remainder of this paper is organized as follows: section 2 motivates the design of the language SES$_{light}$, section 3 describes its syntax and semantics, section 4 formally defines the Confinement problem for SES$_{light}$ APIs, section 5 presents a static analysis procedure for verifying API Confinement, section 6 presents applications of the procedure on certain benchmark examples, 7 describes related work and section 8 concludes.

II. From JavaScript to ES5-strict to SES_{light}

We motivate the design of the language SES_{light} in two steps. We first describe the restrictions ES5S imposes on the form of JavaScript implemented in current browsers and then explain the added restrictions of SES_{light} over ES5S.

A. JavaScript to ES5S

In Dec 2009, the ECMA committee released the 5^{th} edition of the ECMA262 standard [10] which includes a "strict mode" that is approximately a syntactically and semantically restricted subset of the full language. Shifting from normal to strict mode is done by mentioning the "use strict" directive at the beginning of a function body, as in function(){*"use strict"*; .....}. In this paper we analyze the strict mode of ES5 as a separate programming language and assume that all code runs under a global "use strict" directive. Figure 1 summarizes the restrictions enforced by ES5S on JavaScript. The three language properties that hold for ES5S as a result are: *Lexical Scoping*, *Safe Closure-based Encapsulation* and *No Ambient Access to Global Object*. For each of these properties, we briefly explain why they fail for JavaScript and hold for ES5S:

Lexical Scoping. Even though variable bindings in ES3 are almost lexically scoped, the presence of prototype chains on scope objects (or activation records) and the ability to delete variable names makes a static scope analysis of variable names impossible. This makes ordinary renaming of bound variables (α-renaming) unsound and significantly reduces the feasibility of static analysis. For example in the following code, it is impossible to decide the value returned by the call f(), for an arbitrary expression e.

```
Object.prototype[<e>] = 24;
var x = 42;
var f = function foo(){return x;}; f();
```

If the evaluation of expression e returns "x" then the call f() returns 24, else it would return 42. Similar corner cases arise when code can potentially delete a variable name or can use the with construct to artificially insert objects on the scope chain. Recognizing these issues, ES5S forbids deletion on variable names and the use of the with construct. Furthermore, the semantics of ES5S models activation records using the traditional store data structure and therefore without any prototype inheritance.

Safe Closure-based Encapsulation. JavaScript implementations in most browsers support the arguments.caller construct that provides callee code with a mechanism to access properties of the activation object of its caller function. This breaks closure-based encapsulation, as illustrated by the following example: a trusted function takes an untrusted function as argument and checks possession of a secret before performing certain operations.

Restriction	Rationale
No delete on variable names	Lexical Scoping
No prototypes for scope objects	Lexical Scoping
No with	Lexical Scoping
No this coercion	Isolating Global Object
Safe built-in functions	Isolating Global Object
No .callee, .caller on arguments objects	Safe Encapsulation
No .caller, .arguments on function objects	Safe Encapsulation
No arguments and formal parameters aliasing	Safe Encapsulation

Figure 1. ES5S restrictions over JavaScript

```
function trusted(untrusted, secret) {
    if (untrusted() === secret) {
        // process secretObj
    }
}
```

Under standard programming intuition, this code should not leak secret to untrusted code. However the following definition of untrusted would enable it to steal secret.

```
function untrusted() {return arguments.caller.arguments[1];}
```

ES5S eliminates such leaks and make closure-based encapsulation safe by explicitly forbidding implementations from supporting .caller, .arguments on function objects.

No Ambient Access to Global Object. JavaScript provides multiple (and surprising) ways for code to obtain a reference to the global or window object, which is the root of the entire DOM tree and hence security-critical in most setups. For instance, the following program can be used to obtain a reference to the global object.

```
var o = {foo: function (){return this;}}
g = o.foo; g();
```

This is because the this value of a method when called as a function gets coerced to the global object. Further, built-in methods sort, concat, reverse of Array.prototype and valueOf of Object.prototype return a reference to the global object when invoked with certain ill-formed arguments. ES5S prevents all these leaks and only allows access to the global object by using the keyword this in global scope and any host-provided aliases, such as the global variable window.

B. ES5S to SES_{light}

While ES5S simplifies many issues associated with JavaScript, two challenges related to the API Confinement problem remain: (1) All code has *undeniable* write access to the built-in objects, which can be maliciously used to alter the behavior of trusted code that make use of built-in objects, and (2) Code running inside eval is unavailable statically, and so we do not know what global state it accesses. These problems are addressed by the SES_{light} restrictions on ES5S.

The first problem is solved by making all built-in objects, except the global object, *transitively immutable*, which means that all their properties are immutable and the objects cannot be extended with additional properties. Further, all built-in properties of the global object are made immutable.

The second problem is addressed by imposing the restriction that all calls to eval must specify an upper bound on the set of free variables of the code being eval-ed. (Unlike JavaScript, the free variables of a program are statically definable for ES5S; see [35] for a precise definition.) At run-time, the code is evaluated only if its free variables are within the set specified by the arguments. The restricted eval function is called *variable-restricted eval*. For example: the call eval('var x = y + z') is written out as eval('var x = y + z', "y", "z") where {"y", "z"} is the set of free variables. This restriction makes it possible to conservatively analyze eval calls by assuming a worst-case behavior based on the free variables specified.

Like FBJS [36] and the JavaScript subsets devised in previous sandboxing studies [23, 24], SES$_{light}$ does not support setters/getters. However, SES$_{light}$ is a more permissive language subset. For example, SES$_{light}$ allows a form of eval, while the other languages do not. In addition, while SES$_{light}$ has a restricted semantics to support isolation, the corresponding restrictions in FBJS are enforced using a combination of filtering, rewriting and wrapping that is not clearly documented in a public standard. For example, in order to prevent this from referring to the global object, FBJS rewrites the keyword this to ref(this), where ref implements an inlined runtime monitor that does not return the global object. In addition, FBJS does not have full lexical scoping or immutable built-in objects. Since SES$_{light}$ is essentially ES5S without setters/getters, with the variable-restriction on eval and transitively immutable built-in objects, we believe that this clean language design with standardized semantics is more attractive to programmers and developers than previous languages designed to support similar forms of sandoxing and confinement via code rewriting and wrapping.

III. The Language SES$_{light}$

We define the syntax and semantics of SES$_{light}$.

A. Syntax

The abstract syntax of SES$_{light}$ is given in figure 2, using the notation $\tilde{t}$ for a comma-separated list $t_1, \ldots, t_n$. The syntax is divided into values, expressions and statements. A value is either a primitive value, a heap location or one of the error values *TypeError*, *RefError*. Locations include constants for the global object, and all pre-defined built-in objects. Expressions are either variables or values. Statements include assignment, property load, property store, and all representative control flow constructs from ES5S. All statements are written out in a normal form, similar to the A-Normal form of featherweight Java [2]. It is easy to see

Variables and Values

$(Loc)\ l ::=$	$l_g \mid l_{obj} \mid l_{oProt} \mid \ldots$	locations
	$null \mid l_1 \mid \ldots$	
$(PVal)\ pv ::=$	$num \mid str \mid bool \mid undef$	primitives
$(Val)\ v ::=$	$l \mid pv \mid TypeError \mid RefError$	values
$(FVal)\ fv ::=$	function $x(\tilde{y})\{s\}$	function values
$(\mathcal{A})\ a ::=$	$All \mid $Num $\mid \ldots$	annotations
$(Vars^u)\ x, y ::=$	this $\mid$ foo $\mid$ bar $\mid \ldots$	user-variables

Expressions:

$(Exps)\ e ::= \quad x \mid v$

Statements:

$(Stmts^u)\ s, t ::=$	$y = e$	expr
	$y = e_1\ binop\ e_2$	binary expr
	$y = unop\ e$	unary expr
	$y = e_1[e_2, a]$	load
	$e_1[e_2, a] = e_3$	store
	$y = \{x \tilde{:} e\}$	object literal
	$y = [\tilde{e}]$	array literal
	$y = e(\tilde{e_i})$	call
	$y = e[e', a](\tilde{e_i})$	invoke
	$y =$ new $e(\tilde{e_i})$	new
	$y =$ function $x\ (\tilde{z})\{s\}$	function expr
	function $x\ (\tilde{z})\{s\}$	func decl
	eval$(e, \tilde{str})$	eval
	return e	return
	var x	var
	throw e	throw
	$s; t$	sequence
	if (e) then s [else t]	if
	while $(e)\ s$	while
	for $(x\ in\ e)\ s$	forin
	try$\{s_1\}$catch $(x)\{s_2\}$finally$\{s_3\}$	try
	$N \mid Th(v) \mid Ret(v)$	end

Figure 2. Syntax for SES$_{light}$

that using temporary variables, all complex statements from ES5S, except setters/getters and eval, can be re-written into semantics-preserving normalized statements. For example, y = o.f.g.h() can be re-written to $a=o.f ; $b=$a.g ; y=$b.h() with temporary variables $a and $b.

The syntax for property-lookup is augmented with property annotations, which are an optional method to improve the precision of our static analysis method. A property lookup with annotation a is written as $e_1[e_2, a]$. The annotation indicates a bound on the set of string values the expression e_2 can evaluate to. Examples of annotations are: $Num which represents the set {"0", "1", ...}, $Native which represents the sets of built-in method names {"toString", "valueOf", ...} etc. We use the annotation $All to represent the domain of all strings. Using $All, we can trivially translate an un-annotated property lookup to annotated property lookup. We denote the set of all annotations by $\mathcal{A}$ and assume a map $Ann : Str \rightarrow 2^{\mathcal{A}}$ specifying the valid annotations for a given string.

B. Operational Semantics

We define a small step style operational semantics ([32]) for SES$_{light}$, denoted by $(\Sigma, \rightarrow)$. For all expressions and statements except $eval$, the semantics is based on the 5^{th} edition of the ECMA262 standard. In this respect, our semantics is similar to the JavaScript semantics by Maffeis et al [22], which was also based on the ECMA262 standard. The main technical difference in the structure of our semantics and the one by Maffeis et al is that we model scope objects using the standard store data structure and not as first class objects. This simplification was possible due to the more standard scoping semantics of ES5S. The entire semantics is approximately 27 pages long in ASCII, including a model for the DOM and a subset of built-in objects, and is listed online [34]. We now briefly describe the semantics.

Notations Conventions. Loc, Val, $Vars^u$, $Stmts^u$ are the set of all locations, values, user variables and user statements as defined in figure 2. Loc includes l_g which is the (constant) location of the global object. Since the semantics is small step style, it introduces new terms and values in the program state for book-keeping. Such terms and values are called "internal" and are prepended with the symbol '@'. $Vars^@$ and $Stmts^@$ are the sets of all internal variables and statements respectively. $Vars$ is the set of all variables, defined as $Vars^u \cup Vars^@$, and $Terms$ is the set of all terms, defined as $Stmts^u \cup Stmts^@$. For a partial map f, $dom(f)$ is the set of elements on which it is defined. For a value v and partial map f, $f[x \rightarrow v]$ denotes the map obtained by updating the value of $f(x)$ to v.

Heaps, Stacks and States. The complete definitions of Heaps and Stacks are present in figure 3. Stacks contain property records (or activation records) which are partial maps from $Vars$ to the set of records values $RVal$. A record value is either $\perp$, denoting an uninitialized property name or a pair of a value (from Val) and an attributes set specifying whether the property is writable, enumerable or deletable. Unless needed, we will always write records values as values and ignore their attribute part. The empty stack [ ] specifies the global scope. JavaScript supports closures, which are modeled as pairs of statements and stacks, denoting a function's body and lexical scope respectively. Heaps are modeled as partial maps from the set of locations (or object references) to the set of objects. Objects are of two kinds: (1) Non-function objects modeled as property records. (2) Function objects modeled as pairs of property-records and closures. We use $Heaps$, $Stacks$ to denote the domain of heaps and stacks respectively. Finally, a program state is defined as a triple consisting of a heap, a stack and a term. $\Sigma := Heaps \times Stacks \times Terms$ is the domain of all states.

Property Lookup and Variable Resolution. Property lookup and variable resolution for SES$_{light}$ can be defined as functions over a heap and stack. Property lookup uses the prototype-based inheritance mechanism, which is modeled in the semantics using an $@proto$ internal property on all objects that points to their respective prototype object. Given a heap H, the value of property p for a location l is given by the function $Proto(H, l, p)$, defined as follows:

$$\frac{p \notin dom(H(l))}{Proto(H, l, p) = Proto(H, H(l)(\text{`@proto'}), p)}$$

$$\frac{p \in dom(H(l)) \quad v = H(l)(p)}{Proto(H, l, p) = v} \qquad Proto(H, null, p) = undef$$

Variable resolution for SES$_{light}$ is defined in the standard way by traversing down the stack of activation records. It is formalized using the function $Lookup(H, A, x)$, defined as follows

$$\frac{x \in R \quad v = R(x)}{Lookup(H, [R,\ A], x) = v} \qquad \frac{\neg HasProp(H, l_g, x)}{Lookup(H, [\,], x) = RefError}$$

$$\frac{x \notin dom(R)}{Lookup(H, [R,\ A], x) = Lookup(H, A, x)}$$

$$\frac{HasProp(H, l_g, x)}{Lookup(H, [\,], x) = Proto(H, l_g, x)}$$

Here l_g is the reference to the global object and $HasProp(H, l_g, x)$ checks if x appears anywhere on the prototype chain of the global object.

Expression semantics. Semantics of an expression e is given by a map $[\![e]\!] : Heaps \times Stacks \rightarrow Val$, defined as follows:

$[\![x]\!]HA = Lookup(H, A, x)$ $\qquad$ $[\![v]\!]HA = v$

Statement semantics. Semantics of statements are expressed as small-step state transition rules of the form $H, A, t \rightarrow K, B, s$. Rules are divided into axioms and context rules. We define three kinds of termination statements: N for normal completion of execution, $Ret(v)$ for function termination and $Th(v)$ for disrupted execution. For the latter two, v denotes the value returned and thrown respectively. We now explain a few rules to convey the main ideas.

Load. We present the semantics for the annotated load

$Vars^@ := @extensible \mid @class \mid @code \mid @proto \mid @_1, \ldots$
$Attr := writable \mid configurable \mid enumerable$

$Closure := FVal \times Stacks$
$RVal := (Val) \times 2^{Attr} \cup \{\perp\}$
$Records\ R := Vars \rightharpoonup RVal$
$Objects\ o := Records\ \cup\ (Records \times Closure)$
$Stacks\ A, B := [Records^*]$
$Heaps\ H, K := Loc \rightharpoonup Objects$

Figure 3. Heaps and Stacks

statement $y = v_1[v_2, a]$

$$\frac{@_1, @_2 = freshVar() \quad s := @TS(@_2, v_2); @TO(@_1, v_1); y = @_1[@_2, a]}{H, A, y = v_1[v_2, a] \rightarrow H, A, s}$$

$$\frac{a \notin Ann(str)}{H, A, y = l[str, a] \rightarrow H, A, Th(TypeError)}$$

$$\frac{v = Proto(H, l, str) \qquad a \in Ann(str)}{H, A, y = l[str, a] \rightarrow H, A, y = v}$$

The first step is to convert v_2 to a string and v_1 to an object. This is achieved using the internal statements $@TS(@_2, v_2)$ and $@TO(@_1, v_1)$ respectively, where $@_1$ and $@_2$ are internal variables used to store the results of the conversions. Next, if the string value of v_2 matches the annotation a, then the corresponding property is looked up, else a $TypeError$ is thrown.

(Variable-restricted) Eval. We now present the semantics for the variable-restricted eval statement $\mathsf{eval}(str_0, str_1, \ldots, str_n)$. The first step is to convert the argument str_0 to a string, if it is not already in string form. The next step is to parse the string str_0 and check if its free variables are contained in $\{str_1, \ldots, str_n\}$. We refer the reader to [35] for a formal definition of free variables of a SES$_{light}$ term. If the free variable check goes through then a new activation record is placed on the global stack and the parsed term is executed. The reduction rule is as defined follows:

$$\frac{\begin{array}{ll} Parse(str_0) = s & \{str_1, ..., str_n\} \subseteq Free(s) \\ R = NewAR() & A_1 = setVD([R], s) \\ K, B = setFD(H, A_1, s, true) & \end{array}}{H, A, \mathsf{eval}(str_0, str_1, ..., str_n) \rightarrow K, B, @EVAL(s, A)}$$

$setVD$ and $setFD$ methods in the premise scan the eval code for all local variable and function declarations and add the corresponding bindings to the activation record. $@EVAL(\_, A)$ is an internal statement context which reduces any statement placed in it to a terminal value and then restores the stack A in case of normal termination.

Built-in Objects and the DOM. The ES5 standard defines a set of objects and functions that must be initially present in all JavaScript environments. We model this by defining an initial heap and stack H_0, A_0 which contain all the predefined objects. For ease of analysis, we only model a small set of the built-in objects in SES$_{light}$, namely: global object, constructors Object, Array, methods toString, valueOf, hasOwnProperty, propertyIsEnumerable of Object.prototype, methods toString, call, apply of Function.prototype and methods toString, join, concat, push of Array.prototype.

As mentioned in section 2, SES$_{light}$ imposes the restriction that all built-in objects, except the global object, are *transitively immutable*, which means that their $@extensible$ property is set to false and all their properties have attributes *non-configurable* and *non-writable*. Furthermore, all built-in properties of the global object also have the attributes *non-configurable* and *non-writable*.

In addition to the built-in objects, all browser JavaScript environments also contain a pre-defined set of *Document* (DOM) objects whose initial root is pointed to by the 'document' property of the global object. The DOM objects have several properties and methods for manipulating features of the underlying Web page. In this paper, we consider all DOM objects as security-critical and conservatively model all DOM methods using the stub function(x~){return document}, that is, the document object leaks via all methods of all DOM objects.

IV. The API Confinement Problem for SES$_{light}$

In this section, we formally define the confinement problem for SES$_{light}$ APIs. We start by building up the formal machinery required for stating the problem.

A. Preliminaries

Given a state $S := (H, A, t)$, $\mathcal{H}(S)$, $\mathcal{A}(S)$ and $\mathcal{T}(S)$ denote the heap, stack and term part of the state. Given states S and T, we say S reaches T in many steps, denoted by $S \rightsquigarrow T$, iff either $S \rightarrow T$ holds or exists states $S_1, \ldots, S_n$ $(n \geq 1)$ such that $S \rightarrow S_1 \rightarrow \ldots \rightarrow S_n \rightarrow T$ holds. For a state S, $Tr(S)$ is the possibly infinite sequence of states $S, S_1, \ldots, S_n, \ldots$ such that $S \rightarrow S_1 \rightarrow \ldots \rightarrow S_n \rightarrow \ldots$. Given a set of states $\mathcal{S}$, $Reach(\mathcal{S})$ is the set of all reachable states, defined as $\{S' \mid \exists S \in \mathcal{S} : S \rightsquigarrow S'\}$

Labelled Semantics. In the setting considered in this paper, the resources to be confined are references to certain objects that are deemed critical by the hosting page. Since precise references only come into existence once the hosting page code executes, we statically (and conservatively) identify all critical references by their allocation sites in the hosting page code. In order to formalize this, we statically attach labels to all nodes in the syntax tree of a term. For example, the statement if (x) then y = 1 else y = 2 is labelled as $\hat{l}_1$:if (x) then $\hat{l}_2$:y = 1 else $\hat{l}_3$:y=2 where $\hat{l}_i$ are the labels. $\mathcal{L}$ is the domain from which labels are picked. Labels are also attached to heap locations and stack frames, based on the term whose evaluation created them. All rules $H, A, t \rightarrow K, B, s$ are augmented so that any allocated location or activation record carries the label of term t and also any dynamically generated sub-term of s carries the label of term t. Finally, unique labels are attached to all locations on the initial heap H_0. We use $\hat{l}_g$ as the label for the global object.

From here onwards, we will only consider the labelled semantics for SES$_{light}$. To avoid notational overhead, we will use the same symbols l, R and s for labelled locations, activation records and statements and define $Lab(l), Lab(R)$ and $Lab(s)$ respectively as the labels associated with them. The map Lab is naturally extended to sets of heap locations and activation records.

α-Renaming. As discussed earlier, unlike JavaScript, SES_{light} is a lexically scoped language. We formalize this property by defining a semantics preserving procedure for renaming bound variables in a SES_{light} term. The procedure is parametric on a variable renaming map $\alpha : Vars \times \mathcal{L} \rightarrow Vars$ that generates unique names for a particular scope label. The procedure makes use of an auxiliary property named *closest bounding label*, which we define first.

Definition 1: Given a labelled statement s and a variable x appearing in s we define the closest bounding label *of x, denoted by $Bl(x, s)$, as the label of closest enclosing function expression, function declaration or try-catch-finally statement that has x as one of its bound variables.*

We now define the α-Renaming procedure.

Definition 2: [α-Renaming] Given a labelled statement s and a variable renaming map $\alpha : Vars \times \mathcal{L} \rightarrow Vars$, the renamed statement $Rn(s, \alpha)$ is defined by the following procedure: For each variable x appearing in s, if $x \notin Free(s)$, replace x with $\alpha(x, Bl(x, s))$

In order to prove that the above procedure is semantics preserving, we extend the renaming function Rn to labelled program traces and show that renamed and unrenamed traces are *bisimilar*. States are renamed by individually renaming the heap, stack and term components. A heap is renamed by appropriately renaming all closures appearing on it and a stack is renamed by renaming all variables using the label of the property record in which it appears.

Theorem 1: For all wellformed states S, $Rn(Tr(S)) = Tr(Rn(S))$

Proof Sketch: By induction on the length of the trace, with the inductive case proven using a case analysis on the set of reduction rules. □

B. Problem Definition

In this section, we formally state the API Confinement problem. We assume that the security-critical resources are specified using a set of forbidden allocation-site labels P. Further, we assume that labels of all DOM objects also belong to the set P. This is not required for the correctness of the subsequent analysis, but is the special case under which our conservative model of the DOM is practically relevant.

In accordance with the API+Sandbox mechanism, the hosting page code runs first and creates an API object, which is then handed over to the untrusted code that runs next. The hosting page code is called the *trusted API service*. We assume for simplicity that the hosting page stores the API object in some shared global variable api. In order for this mechanism to be secure, untrusted code must be appropriately restricted so that the only trusted code global variable it has access to is api. Using the variable-restricted SES_{light} eval, it is straightforward to restrict any term s to any specific set of global variables $\{x_1, \ldots, x_n\}$ simply by rewriting s to $\text{eval}(s, x_1, \ldots, x_n)$.

In order to set up the confinement problem we also provide untrusted code access to a global variable un, which is used as a *test variable* in our analysis and is initially set to undefined. The objective of untrusted code is to store a reference to a forbidden object in it. Without loss of generality, we assume that the API service t is suitably-α-renamed according to the procedure in definition 2 so that it does not use the variable un.

In summary, if t is the trusted API service and s is the untrusted code then the overall program that executes in the system is $t; \text{var un}; \text{eval}(s, \text{“api”}, \text{“un”})$. Informally, the API confinement property can be stated as: for all terms s, the execution of $t; \text{var un}; \text{eval}(s, \text{“api”}, \text{“un”})$ with respect to the initial heap-stack H_0, A_0 never stores a forbidden object in the variable un. We now formally define this property. The definition makes use of the map $PtsTo : Vars^u \times 2^{\Sigma} \rightarrow 2^{\mathcal{L}}$ which we define first. Recall that l_g is the location of the global object.

Definition 3: [Points-to] Given a set of states $\mathcal{S} \in 2^{\Sigma}$, and a variable $v \in Vars^u$, $PtsTo(l, v, \mathcal{S})$ is defined as the set: $\{Lab(H(l_g)(v)) \mid \exists A, t : H, A, t \in \mathcal{S}\}$

Given a trusted API service, let $\mathcal{S}_0(t)$ be the set of states $\{H_0, A_0, t; \text{var un}; \text{eval}(s, \text{“api”}, \text{“un”}) \mid s \in \text{SES}_{light}\}$.

Definition 4: [Confinement Property] A trusted service t safely encapsulates a set of forbidden allocation-site labels P iff $PtsTo(\text{“un”}, Reach(\mathcal{S}_0(t))) \cap P = \emptyset$. We denote this property by $Confine(t, P)$.

API Confinement Problem. *Given a term t and a set of forbidden allocation-site labels P, verify $Confine(t, P)$*

V. Analysis Procedure

In this section we define a procedure $\mathcal{D}(t, P)$ for verifying that an API service t safely confines a set of critical resources P. The main idea is to define a tractable procedure for over-approximating the set $PtsTo(\text{“un”}, Reach(\mathcal{S}_0(t)))$ where $\mathcal{S}_0(t) = \{H_0, A_0, t; \text{var un}; \text{eval}(s, \text{“api”}, \text{“un”}) \mid s \in \text{SES}_{light}\}$. We adopt an *inclusion-based*, *flow-insensitive* and *context-insensitive* points-to analysis technique [1] for over-approximating this set. This is a well-studied and scalable points-to analysis technique. Flow-insensitivity means that the analysis is independent of the ordering of the statements and context-insensitivity means that the analysis only models a single activation record for each function, which is shared by all call-sites. Given the presence of closures and absence of a static call graph in JavaScript, a context-sensitive analysis is known to be significantly more expensive than a context insensitive one (see [14, 27] for complexity results). The technique adopted in this paper on the other-hand is polynomial time. Given that there has been very little prior work (see [13]) on defining provable-sound

static analyses for JavaScript, we believe that a provably-sound flow-insensitive and context-insensitive analysis is a reasonable first step.

In adopting the well-known *inclusion-based* based flow and context insensitive points-to analysis technique to our problem, we are faced with the following challenges: (1) Statically encoding eval statements (2) Statically reasoning about the entire set of states $\mathcal{S}_0(t)$ at once. (3) Correct modeling of the various non-standard features of the SES$_{light}$ semantics.

We resolve these challenges as follows. As discussed earlier, the arguments to eval in SES$_{light}$ statically specify a bound on the set of free variables of the code being eval-ed. We use this bound to define a worst case encoding for eval, which essentially amounts to creating all possible points-to relationships between all the objects reachable from the set of free variables. Since the encoding only depends on the set of free variables and is independent of the actual code being evaluated, it resolves both challenges (1) and (2). For (3), we leverage upon the insights gained while developing the formal semantics for SES$_{light}$ and formulate our abstractions in a sound manner. We also present a proof of correctness for our procedure which guarantees that we (conservatively) respect the semantics.

We now present the details of the procedure. We follow the approach of Whaley et al. [40] and express our analysis algorithm in Datalog. Before describing the details of this approach, we provide a quick introduction to Datalog.

Quick introduction to Datalog. A Datalog program consists of *facts* and *inference rules*. Facts are of the form $P(t_1, \ldots, t_n)$ where P is a predicate symbol and t_i are terms, which could be constants or variables. Rules are sentences that provide a means for deducing facts from other facts. Rules are expressed as *horn clauses* with the general form $L_0 :- L_1, \ldots, L_n$ where L_i are facts. Given a set of facts $\mathcal{F}$ and a set of inference rules $\mathcal{R}$, $Cons(\mathcal{F}, \mathcal{R})$ is the set of all "consequence" facts that can be obtained by successively applying the rules to the facts, upto a fixed point. As an example if $\mathcal{F} := \{edge(1,2), edge(2,3)\}$ and

$$\mathcal{R} := \left\{ \begin{array}{l} path(x,y) :- \ edge(x,y); \\ path(x,z) :- \ edge(x,y), path(y,z) \end{array} \right\}$$

then $Cons(\mathcal{F}, \mathcal{R})$ is the set $\{edge(1,2), edge(2,3), path(1,2), path(2,3), path(1,3)\}$. We refer the reader to [7] for a comprehensive survey of Datalog and its semantics.

Procedure Overview. A high-level overview of the procedure $\mathcal{D}(t, P)$ is as follows:
(1) Collect facts (expressed over Datalog relations) about the statements present in t and add them to a Database. Add facts about the initial heap H_0 and the term var un; eval$(s, \text{"un"}, \text{"api"})$ for *any* $s \in$ SES$_{light}$.
(2) Conservatively encode the semantics of SES$_{light}$ in the form Datalog inference rules and then use them to compute the consequence set of the Database obtained in (1).
(3) Analyze the databases from (1) and (2), and check for confinement violating facts.

The rest of this section is organized as follows: 5.1 describes the encoding of SES$_{light}$ statements as Datalog facts, 5.2 presents the inference rules, 5.3 presents the formal definition of the procedure and 5.4 provides a soundness argument.

A. Datalog Relations and Encoding

Our encoding of program statements into Datalog facts, makes use of the standard abstraction of heap locations as allocation-site labels. Since JavaScript represents objects and function closures in the same way, this applies to function closures as well. In the terminology of control-flow analysis, this abstraction makes our analysis 0-CFA. Further, the analysis only supports *weak updates*, which means we aggregate values with each variable and property assignment.

Facts are expressed over a fixed set of relations $\mathbb{R}$, enumerated in figure 4 along with their domains. $V \subseteq Vars$ is the domain for variable and field names, $L \subseteq \mathcal{L}$ is the domain for allocation-site labels (abstract locations) and I is the domain for function argument indices. A similar set of relations has been used for points-to analysis of Java in [3, 40]. Besides relations that capture facts about the program, we use $Heap, Stack, Prototype$ to capture facts about the heap and stack. $Heap(\hat{l}_1, x, \hat{l}_2)$ encodes that an object with label $\hat{l}_1$ has a field x pointing to an object with label $\hat{l}_2$ and $Stack(x, \hat{l})$ encodes that variable x points to an object with label $\hat{l}$. $Prototype$ captures the prototype-inheritance relation between various objects.

We define $Facts$ as the set of all possible facts that can be expressed over the relations in $\mathbb{R}$. We now describe the encoding of statements in SES$_{light}$ into facts over $\mathbb{R}$. For each label $\hat{l}$, we assume a unique and countably-infinite set of labels $h(\hat{l}, 1), h(\hat{l}, 2), \ldots$ associated with it. The purpose of these labels is to denote objects that get created *on the fly* during the execution of a statement. Further we use a variable renaming map $\alpha : Vars \times \mathcal{L} \rightarrow Vars$ in defining our encoding. The encoding of a statement s depends on the label $\hat{l}$ of the nearest enclosing scope in which it appears and is given by the map $Enc_{\mathcal{T}}(s, \hat{l})$. Due to space limitations we only describe the main ideas here and present the formal definition of $Enc_{\mathcal{T}}(s, \hat{l})$ in the accompanying tech report [35].

Binary Expression Statement. According to the semantics of a binary operation statement $s := y = x_1 \ binop \ x_2$, if $binop \in \{\$\$, ||\}$ and if x_1 or x_2 resolve to an object then they could potentially get assigned to y. We therefore conservatively encode such statements by $\{Assign(y, x_1), Assign(y, x_2)\}$. This is a subtle semantic

feature that existing JavaScript points-to analysis frameworks [11, 15] don't seem to account for. Furthermore, if $binop \notin \{\$\$, ||\}$ and if x_1 or x_2 resolve to an object, then the evaluation might trigger an implicit 'ToPrimitive' type conversion which could potentially invoke the $valueOf$ and $toString$ methods of the object. We encode such statements by $\{TP(x_1, \hat{l}), TP(x_2, \hat{l})\}$, where $TP(x, \hat{l})$ encodes that a 'ToPrimitive' conversion should be triggered on variable x in scope $\hat{l}$.

Load. The evaluation of a load statement $s := y = x_1[x_2, a]$ could potentially involve a 'ToPrimitive' conversion on the argument x_2 and a 'ToObject' conversion on the object x_1. The statement is encoded as

$$\{TP(x_2, \hat{l}), Stack(x_1, \hat{l}_1), ObjType(\hat{l}_1), NotBuiltin(\hat{l}_1)\} \bigcup \{Load(y, x_1, a)\}$$

Here $\hat{l}_1 = h(Lab(s), 1)$ is the abstract location of the object created on the fly from the 'ToObject" conversion. The first set in the union encodes that x_1 points to a non-built-in object with abstract location $\hat{l}_1$ and that x_2 must be converted to a primitive value in the scope $\hat{l}$. $Load(y, x_1, a)$ encodes that contents of $x_1.p$ flow into y for all property names p that annotate to a.

Function Declaration and Calls. A function declaration $s :=$ function $x(\tilde{y})\{s_1\}$ is encoded as:

$$\left\{ \begin{array}{l} FormalArg(\hat{l}_1, 1, y_1), \ldots, FormalArg(\hat{l}_1, n, y_n), \\ FormalArg(\hat{l}_1, \text{“}a\text{”}, \text{arguments}), FuncType(\hat{l}_1), \\ FormalArg(l_1, \text{“}t\text{”}, \text{this}), Stack(x, \hat{l}_1) \\ ObjType(\hat{l}_2), Heap(\hat{l}_1, \text{“}prototype\text{”}, \hat{l}_2) \end{array} \right\} \bigcup Enc_{\mathcal{T}}(s_1, \hat{l}_1)$$

Here $\hat{l}_1 = h((Lab(s), 1)$ and $\hat{l}_2 = h((Lab(s), 2)$ are abstract locations for the function and prototype objects that get created dynamically. $FormalArg$ encodes the positions of all the formal arguments, including default arguments this and arguments, whose positions are denoted by "t" and "a" respectively.

A function call statement $y := x(\tilde{x_i})$ is similarly encoded using facts of the form $Actual(x, i, x_i, y, \hat{l})$ where x_i is the actual argument at position i, y is the return variable and $\hat{l}$ is the label of the nearest enclosing scope.

(Variable-restricted) Eval. The evaluation of a variable-restricted eval statement $s :=$ eval$(x, \tilde{str})$ forces the free variables of the code being eval-ed to be contained in $\{\tilde{str}\}$. Since we do not know the code statically, we conservatively assume that all possible points-to relationships are created between all objects reachable from the free and bound variables. To make the encoding finite, we summarize all the bound variables by a single variable $\alpha(\text{“}x_{eval}\text{”}, Lab(s))$ (here "x_{eval}" is an arbitrarily picked variable name) and all locally allocated objects by a single abstract location $\hat{l}_1 = h(Lab(s), 1)$. For the enclosing scope $\hat{l}$, the encoding is given by the set $Eval(\hat{l}, \hat{l}_1, \alpha(\text{“}x_{eval}\text{”}, Lab(s)), \tilde{str})$, defined formally in figure 5. The set is obtained by instantiating all relations with all possible valid combinations of the variables in $\{\alpha(\text{“}x_{eval}\text{”}, Lab(s)), \tilde{str}\}$ and locations in $\{\hat{l}, \hat{l}_1\}$.

Relations for encoding programs:

$$\begin{array}{ll} Assign : 2^{V \times V} & Throw : 2^{L \times V} \\ Load : 2^{V \times V \times V} & Catch : 2^{L \times V} \\ Store : 2^{V \times V \times V} & Global : 2^{V} \\ FormalArg : 2^{L \times I \times V} & Annotation : 2^{V \times V} \\ FormalRet : 2^{L \times V} & ObjType : 2^{L} \\ Instance : 2^{L \times V} & FuncType : 2^{L} \\ ArrayType : 2^{L} & NotBuiltin : 2^{L} \\ Actual : 2^{V \times I \times V \times V \times L} & \end{array}$$

Relations for encoding the heap-stack:

$$\begin{array}{ll} Heap : 2^{L \times V \times L} & Stack : 2^{V \times L} \\ Prototype : 2^{L \times L} & \end{array}$$

Figure 4. Datalog Relations

Built-in Objects and DOM. We encode all built-in objects and DOM objects present on the initial heap H_0 as a set of facts and rules $\mathcal{I}_0$. For all objects references l_1, l_2 and properties x such that such that $H_0(l_1)(x) = l_2$, $\mathcal{I}_0$ contains the fact $Heap(Lab(l_1), x, Lab(l_2))$. For each built-in method, $\mathcal{I}_0$ contains appropriate rules over $Actual$ facts that capture the semantics of the method. We give the rules for the Function.prototype.apply method, labeled by $\hat{l}_{apply}$, as an example.

According to the semantics of the apply method, the call x_0.apply(x_1, x_2) involves calling the function pointed to by x_0 with this value x_1 and arguments as stored on the array x_2. It is encoded as follows:

$$\begin{array}{ll} Actual(x_0, \text{“}t\text{”}, x_1, y, \hat{l}_{apply}) : - & \text{[APPLY1]} \\ \quad Actual(x, \text{“}t\text{”}, x_0, y, \hat{l}_1), Actual(x, 1, x_1, y, \hat{l}_1), Stack(x, \hat{l}_{apply}) & \\ Actual(x_0, i, x_3, y, \hat{l}_{apply}) : - & \text{[APPLY2]} \\ \quad Actual(x, \text{“}t\text{”}, x_0, y, \hat{l}_1), Actual(x, 2, x_2, y, \hat{l}_1), & \\ \quad Heap(x_2, \$Num, x_3), Stack(x, \hat{l}_{apply}) & \end{array}$$

Encoding built-in methods using rules provides much better call-return matching than the naive encoding using $FormalArg$ facts. This turned out to be very useful in our experiments as calls to built-in methods are pervasive in most API definitions. For all built-in prototype objects, $\mathcal{I}_0$ contains rules for capturing the inheritance relation. For example, the following rule is used for the Object.prototype object which is labelled as $\hat{l}_{oProt}$.

$$Prototype(\hat{l}, \hat{l}_{oProt}) : - ObjType(\hat{l})$$

DOM methods are encoded by encoding the function declaration function$(\tilde{x})\{$return document$\}$.

B. Inference Rules

We now briefly describe the set of inference rules $\mathcal{R}$, which model a flow and context insensitive semantics of SES$_{light}$. The rules are formally defined in figure 6. Since

$Eval(l_{outer}, l_{local}, x_0, \tilde{x})$ is formally defined as:

$$\begin{array}{l}\{Assign(v_1, v_2) \mid v_1, v_2 \in V\} \cup \\ \{Load(v_1, v_2, \text{“\$All”}) \mid v_1, v_2 \in V\} \cup \\ \{Store(v_1, \text{“\$All”}, v_2) \mid v_1, v_2 \in V\} \cup \\ \{Actual(v_1, i, v_2, v_3, l) \mid v_1, v_2, v_3 \in V; l \in L\} \cup \\ \{FormalArg(l_{local}, i, v) \mid v \in V\} \cup \\ \{FormalRet(l_{local}, v) \mid v \in V\} \cup \\ \{Instance(l_{local}, v) \mid v \in V\} \cup \\ \{Throw(l, v) \mid v \in V; l \in L\} \cup \\ \{Catch(l, v) \mid v \in V; l \in L\} \cup \\ \{NotBuiltin(l_{local})\} \cup \\ \{FuncType(l_{local})\} \cup \\ \{ArrayType(l_{local})\} \cup \\ \{ObjType(l_{local})\}\end{array}$$

where $V := \{x_0, \tilde{x}\}, L := \{l_{local}, l_{outer}\}$

Figure 5. Encoding Eval Statements

it is clear from the context, we elide the hat and use symbols l, m, n and k for labels.

Assign, Load and Store. Rules [ASSIGN], [LOAD] and [STORE1] are straightforward and model the semantics of assignments, load and store statements. Rules [PROTOTYPE1] and [PROTOTYPE2] conservatively flatten all prototype chains by taking the reflexive and transitive closure of the relation $Prototype$. Rules [STORE2] and [STORE3] capture that an annotated property store gets reflected on all the concrete property names that satisfy the annotation.

ToPrimitive. Rules [TP1] and [TP2] model the semantics of 'ToPrimitive' conversion. Given a fact $TP(x, l)$, the rule derives a call to the 'toString' and 'valueOf' methods of all objects stored at x. Since the value returned by a 'ToPrimitive' conversion is primitive, it is discarded by specifying a the internal variable $\$dump$ as the return variable.

Function Calls. Function calls are handled by rules [ACTUAL1], [ACTUAL2] and [ACTUAL3]. Since functions are modelled as objects in JavaScript, call targets are also resolved via the heap and stack. The rule [ACTUAL1] flows actual parameters to formal parameters, [ACTUAL2] flows formal return values to actual return values and [ACTUAL3] propagates "throws" across the call chain.

Global and Catch Variables. Since global variables are properties of the global object, assignments to global variables are reflected on the global object and vice versa. This is modeled by rules [GLOBAL1] and [GLOBAL2]. The rule [CATCHVAR] conservatively flows 'throws' from a particular scope into all 'catch' variables appearing in that scope.

C. Procedure for Verifying API Confinement

The procedure $\mathcal{D}(t, P)$ for verifying that API service t confines a set of allocation-site labels P is defined in figure 7. It uses the global object label $\hat{l}_g$ and an abstract points-to map $PtsTo_{\mathcal{D}} : Vars^u \times 2^{Facts} \rightarrow 2^{\mathcal{L}}$ defined as follows.

$$Stack(x, l) :- Stack(y, l), Assign(x, y) \quad \text{[ASSIGN]}$$

$$Stack(x, n) :- \quad \text{[LOAD]}$$
$$Load(x, y, f), Prototype(l, m),$$
$$Heap(m, f, n), Stack(y, l)$$

$$Heap(l, f, m) :- \quad \text{[STORE1]}$$
$$Store(x, f, y), Stack(x, l), NotBuiltin(l), Stack(y, m)$$

$$Store(x, a, y) :- \quad \text{[STORE2]}$$
$$Store(x, f, y), Annotation(f, a)$$

$$Store(x, f, y) :- \quad \text{[STORE3]}$$
$$Store(x, a, y), Annotation(f, a)$$

$$Annotation(f, \text{“\$All”}) \quad \text{[ANNOTATION]}$$

$$Actual(n, \text{“}t\text{”}, x, \text{“\$}dump\text{”}, k) :- \quad \text{[TP1]}$$
$$TP(x, k), Stack(x, l), Prototype(l, m),$$
$$Heap(m, \text{“}toString\text{”}, n), FuncType(n)$$

$$Actual(n, \text{“}t\text{”}, x, \text{“\$}dump\text{”}, k) :- \quad \text{[TP2]}$$
$$TP(x, k), Stack(x, l), Prototype(l, m),$$
$$Heap(m, \text{“}valueOf\text{”}, n), FuncType(n)$$

$$Assign(y, z) :- \quad \text{[ACTUAL1]}$$
$$Actual(f, i, z, x, k), Stack(f, l), FormalArg(l, i, y)$$

$$Assign(x, y) :- \quad \text{[ACTUAL2]}$$
$$Actual(f, i, z, x, k), Stack(f, l), FormalRet(l, y)$$

$$Throw(k, x) : - \quad \text{[ACTUAL3]}$$
$$Actual(f, i, y, z, k), Stack(f, l), Throw(l, x)$$

$$Prototype(l, l) \quad \text{[PROTOTYPE1]}$$

$$Prototype(l, n) :- \quad \text{[PROTOTYPE2]}$$
$$Prototype(l, m), Prototype(m, n)$$

$$Prototype(l, q) :- \quad \text{[PROTOTYPE3]}$$
$$Instance(l, y), Stack(y, m),$$
$$Prototype(m, n), Heap(n, \text{“}prototype\text{”}, q)$$

$$Heap(\hat{l}_g, f, l) :- Stack(f, l), Global(f) \quad \text{[GLOBAL1]}$$

$$Stack(f, l) :- Heap(l_g, f, l) \quad \text{[GLOBAL2]}$$

$$Assign(x, y) :- Catch(k, x), Throw(k, y) \quad \text{[THROW]}$$

Figure 6. The set of Inference Rules $\mathcal{R}$

Definition 5: [Abstract Points-to] Give a set of facts $\mathcal{F} \in 2^{Facts}$ and a variable $v \in Vars^u$, $PtsTo_{\mathcal{D}}(v, \mathcal{F})$ is defined as $\{\hat{l} \mid Stack(v, \hat{l}) \in \mathcal{F}\}$

The first step of the procedure is to pick any program s and encode the term t; var "un"; eval(s, "un", "api") in global scope. Given the way eval statements are encoded, the encoding of the above term does not depend of the term s. The next step is to compute the set of all possible consequences of the encoded facts, under the inference rules $\mathcal{R}$ defined in

Procedure $\mathcal{D}(t, P)$:

1) Pick any term $s \in \text{SES}_{light}$ and compute $\mathcal{F}_0(t) = Enc_{\mathcal{T}}(t; \mathsf{var\ un}; \mathsf{eval}(s, \text{“api”}, \text{“un”}), \hat{l}_g) \cup \mathcal{I}_0$.
2) Compute $\mathcal{F} = Cons(\mathcal{F}_0(t), \mathcal{R})$.
3) Show that $PtsTo_{\mathcal{D}}(\text{“un”}, \mathcal{F}) \cap P = \emptyset$.

Figure 7. Procedure for Verifying $Confine(t, P)$

figure 6. The final step is to compute the abstract points-to set of the variable un over this consequence set and check if it contains any labels from the set P. Since the maps $Enc_{\mathcal{T}}$, $Cons$ and $PtsTo_{\mathcal{D}}$ are computable, the procedure is decidable. The procedure is listed purely from the correctness standpoint and does make any efficiency considerations.

D. Soundness

We now prove soundness of the procedure $\mathcal{D}(t, P)$ by showing that for all terms t and allocation-site labels P, $\mathcal{D}(t, P) \implies Confine(t, P)$. Our proof is very close to the one given by Midtgaard et al. in [26] for soundness of 0-CFA analysis. The crux of the proof is in defining a map $Enc : 2^{\Sigma} \rightarrow 2^{Facts}$ (abstraction map) for encoding a set of program states as a set of Datalog facts, and showing that the for any set of states, the set of consequence facts safely over-approximates the set of reachable states, under the encoding.

Encoding of States. We rigorously define the encoding of states in [35] and present only the main ideas here. States are encoded by separately encoding the heap, stack and term. Terms are encoded using the map $Enc_{\mathcal{T}}$ and stacks are encoded by collecting all facts of the form $Stack(x, Lab(l))$ such that variable x stores location l on the stack. Heaps are encoded by collecting all facts of the form $Heap(Lab(l_1), x, Lab(l_2))$ such that property x of location l_1 stores location l_2, and additionally encoding all function-closures (using the term and stack encoding) that are present on the heap.

Results. Our first result is that for a set of states $\mathcal{S}$, the encoding of the set of all states reachable from $\mathcal{S}$, is over-approximated by the set of all consequence facts derivable from the encoding of $\mathcal{S}$.

Lemma 1: Let $\mathcal{R}$ be the inference rules defined in figure 6. For all set of states $\mathcal{S} \in 2^{\Sigma}$, $Enc(Reach(\mathcal{S})) \subseteq Cons(Enc(\mathcal{S}), \mathcal{R})$

Proof Sketch: Given an element $\mathcal{S} \in 2^{\Sigma}$, we define the concrete single-step evaluation map $N_{\rightarrow}(\mathcal{S})$ as $\mathcal{S} \cup \{S' \mid \exists S \in \mathcal{S} : S \rightarrow S'\}$. It is easy to see that $Reach(\mathcal{S})$ is the smallest fixed point of $N_{\rightarrow}$ above $\mathcal{S}$, in the powerset lattice 2^{Σ}.
Given an element $\mathcal{F} \in 2^{Facts}$, we define the abstract single-step evaluation map $N_{\mathcal{D}}(\mathcal{F})$ as $\mathcal{F} \cup Infer_1(\mathcal{F}, \mathcal{R})$ where $Infer_1(\mathcal{F}, \mathcal{R})$ is the set of facts obtained by applying the rules $\mathcal{R}$ exactly once[1]. Under the Herbrand semantics of Datalog, $Cons(\mathcal{F}, \mathcal{R})$ is the smallest fixed point of $N_{\mathcal{D}}$ above $\mathcal{F}$, in the powerset lattice 2^{Facts}
Next, we show by an induction on the set of reduction rules that for all $\mathcal{S} \in 2^{\Sigma}$, there exists $n \geq 1$ such that: $Enc(N_{\rightarrow}(\mathcal{S})) \subseteq N^n_{\mathcal{D}}(Enc(\mathcal{S}))$
It is straightforward to prove the lemma from this property. □

Recall the set of initial states $\mathcal{S}_0(t)$ and initial facts $\mathcal{F}_0(t)$ from the definitions of $Confine(t, P)$ and $\mathcal{D}(t, P)$ respectively. Our next result shows that the $\mathcal{F}_0(t)$ over-approximates the encoding of $\mathcal{S}_0(t)$.

Lemma 2: For all terms $t \in SES_{light}$, $Enc(\mathcal{S}_0(t)) \subseteq \mathcal{F}_0(t)$

The proof is straightforward and follows from the definition of Enc. The final lemma for proving soundness is that the abstract points-to map $PtsTo_{\mathcal{D}}$ safely over-approximates the concrete points-to map, under the encoding.

Lemma 3: For all $v \in Vars^u$ and set of states $\mathcal{S} \in 2^{\Sigma}$, $PtsTo(v, \mathcal{S}) \subseteq PtsTo_{\mathcal{D}}(v, Enc(\mathcal{S}))$.

The proof is straighforward and follows from the definitions of Enc, $PtsTo$ and $PtsTo_{\mathcal{D}}$. We now state the soundness theorem.

Theorem 2: [Soundness] For all terms t and forbidden allocation-site labels P, $\mathcal{D}(t, P) \implies Confine(t, P)$

Proof Sketch: From figure 7, $\mathcal{D}(t, P)$ holds iff $PtsTo_{\mathcal{D}}(\text{“un”}, Cons(\mathcal{F}_0(t), \mathcal{R})) \cap P = \emptyset$. From monotonicity of $Cons$ and $PtsTo_{\mathcal{D}}$ and lemmas 1, 2, 3, it follows that the set $PtsTo(\text{“un”}, Reach(\mathcal{S}_0(t)))$ is a subset of $PtsTo_{\mathcal{D}}(\text{“un”}, Cons(\mathcal{F}_0(t), \mathcal{R}))$. The theorem follows immediately from this result. □

VI. Applications

In this section, we demonstrate the value of our analysis procedure by analyzing three benchmark examples: Yahoo! ADsafe library [9], the Sealer-Unsealer mechanism ([17, 33]) and the Mint mechanism [30]. All these examples are of APIs that have been designed with an emphasis on robustness and simplicity, and have been previously subjected to security analysis. We analyze these examples under the semantics and threat model of SES_{light}. The goal of our experiments was to test the effectiveness of the procedure $\mathcal{D}(t, P)$ by checking if it could correctly prove confinement properties for these well-studied APIs.

Analyzer Architecture. We implemented the procedure $\mathcal{D}(t, P)$ from figure 7 in the form of a tool named ENCAP. The tool has a JavaScript parser at the front end and the bddbddb Datalog engine [39] at the back end. Given an input API definition and a list of precious creation-site

[1]Also known as the *elementary production principle* (see [7])

labels, the parser generates an SES$_{light}$ AST which is then encoded into a set of Datalog facts. As described in the procedure, this encoding is combined with the encoding of the initial heap and the encoding of the eval statement var "un"; eval(s, "api", "un") for any $s \in$ SES$_{light}$.

Running SES$_{light}$ on an ES5S browser. The procedure $\mathcal{D}(t, P)$ is designed to verify confinement of APIs written in SES$_{light}$, under the SES$_{light}$ threat model. The ideal deployment scenario would be for browsers that primitively support SES$_{light}$. Given the absence of such browsers, we present a first cut to an approach for emulating the SES$_{light}$ restrictions on a browser supporting ES5S. The main idea is to run an initialization script that makes the heap compliant with the initial SES$_{light}$ heap and then use a static verifier on all code that runs subsequently. The goal of the static verifier is to ensure that the code is valid SES$_{light}$ code and that it does not use any $-prefixed variable names, which is a namespace reserved for book-keeping purposes. For the sake of emulation, we modify the syntax of annotated property lookups from $e_1[e_2, a]$ to $e_1[a(e_2)]$, that is, annotations are expressed as (dynamic type-checking) functions applied on the property being accessed.

The initialization script performs the following steps: (1) Makes all built-in objects and properties that are *not* modeled in SES$_{light}$ unreachable from the ones that are modeled. This can be done using the delete $e_1[e_2]$ construct. (2) Replaces the built-in eval function with a wrapper that uses an SES$_{light}$ parser, written in ES5S, to ensure that code being evaled has all its free variables mentioned as arguments, and that no object literals appearing in the code contain literal get and set properties. The latter ensures that code does not use setters and getters. (3) Makes all built-in objects, except the global object, *transitively immutable*, by applying the built-in method Object.freeze to them, which results in making the objects non-extensible and all their properties non-configurable and non-writable. (4) For each annotation a, we define a non-configurable and non-writable property named a on the global object (using Object.defineProperty), and store a annotation-checking function on it. The code for the function is as follows.

```
var a = function(x){ var $= String(x);
          if(Ann($,a){return $}] else{throw "bad"}}
```

Here Ann(m,n) is a pure function that checks if string m annotates to string n. Recall that annotations in SES$_{light}$ are $-prefixed and therefore the properties created would not be tampered or shadowed by code running subsequently.

We have an implementation of the initialization script described above, but we do not have any rigorous proof of correctness for it yet. We conjecture that for all SES$_{light}$ terms t that do not use $-prefixed variable names, the execution of t on the initial SES$_{light}$ heap and stack under the SES$_{light}$ semantics, is safely emulated by the execution of t on the appropriately initialized ES5S heap and stack under the ES5S semantics.

A. ADsafe

Our first application is the Yahoo! ADsafe framework defined by Douglas Crockford [9] for protecting host pages from untrusted advertisements that contain arbitrary JavaScript code. Following the API+Sandbox approach, the framework has two main components: (1) An ADsafe library that provides restricted access to the DOM and other global variables. (2) A static filter JSLint that discards untrusted JavaScript code if it makes use of certain language constructs like this, eval, with or properties beginning with "___" etc. The goal of the filter is to ensure that JavaScript code that passes through it only accesses security-critical objects by invoking methods on the ADsafe library.

As described in this paper, under the SES$_{light}$ semantics the JSLint goal can be achieved simply by restricting all untrusted code to the SES$_{light}$ subset and wrapping it with the context eval(_,"api"), where api stores the ADsafe library object. In our experiments, we analyze if the ADsafe library confines the DOM object, under the SES$_{light}$ semantics and threat model. Although the ADsafe library was implemented in JavaScript, it does not use setters/getters and eval, and can be de-sugared (using temporary variables) into semantically equivalent SES$_{light}$ code, thus making it amenable to confinement analysis using ENCAP.

Adding Annotations. In order to make our analysis precise and to support certain JSLint restrictions on untrusted code, we add suitable property annotations to the ADsafe library implementation and to the encoding of eval statements. The ADsafe library reserves a set of property names to hide security-critical objects and certain book-keeping information. This set of property names is blacklisted and JSLint filters out all untrusted programs that name any property from this set. We support this restriction in our analysis by annotating all *Load* and *Store* facts in the encoding of eval statements with the annotation $Safe which ensures that the property name is not blacklisted. The annotation $Safe is also added to patterns of the form if (!reject(name)){ ... object[name] ... } in the library implementation, where reject is a function that checks if name is blacklisted. The other annotation used in the library implementation is $Num, which is added in the context of for-loops to property lookups involving the loop index.

Attack. We ran ENCAP on the ADsafe library (approx. 1700 loc) and found that it leaks the document object via the methods lib and go. The running time of the analysis was 5 mins 27 secs on a standard workstation. After analyzing the methods lib and go, we were able to construct an actual client program that used these methods to directly access the document object, thus confirming the leak to be a true positive. The exploit code is present in figure 8.

In order to explain the root cause of the attack, we describe the methods go and lib. The method go(id,f) takes

```
ADSAFE.lib(
      "___nodes___",
      function(lib){
          var o = [{appendChild:
                        function(x){var steal = x.ownerDocument)},
                    tagName:1}];
          return o;}
);
// sets adsafe_lib.___nodes___ to o
ADSAFE.go(
     "test",
      function(dom,lib){
          // lib points to the adsafe_lib object
          var frag = dom.fragment();
          var f = frag.value;
          // f points to the value method of the dom library
          lib.v = f;
          lib.v(); }
);
```

Figure 8. ADsafe exploit code

a function f as an argument and provides it with an object named dom that has methods that wrap the original DOM methods, and a certain library object adsafe_lib that is meant to store libraries defined by untrusted code. The adsafe_lib object method is populated by the lib method which is defined as function (name, f){adsafe_lib[name] = f(adsafe_lib); One of the confinement mechanisms used in the ADsafe library is to hide DOM objects in the "___" property of certain objects and forbid untrusted code from writing to or reading from "___" properties. This mechanism is broken by the lib method which allows untrusted code to write to "___" property of the adsafe_lib object, thus leading to the attack. We refer the reader to [35] for further details.

Fixing the Attack. A fix for the attack is to rewrite the lib method using the annotation $Safe in the following way.

```
function (name, f){if(!reject_name(name)){
                    adsafe_lib[name, $Safe] = f(adsafe_lib);}
```

With this rewriting, ENCAP reports no DOM leaks, thus proving that the ADsafe library safely confines the DOM object under the added annotations and the SES_{light} threat model. We reported the vulnerability to Yahoo! and the corresponding fix was adopted immediately.

B. Sealer-Unsealer Pairs

Our next example is an implementation of the Sealer-Unsealer encapsulation technique, which was first introduced by Morris [17] in 1973, for providing an encryption decryption like mechanism for functions. Sealer-Unsealer pairs are important security mechanisms used in designing capability based systems. We analyzed an SES_{light} implementation of sealers and unsealers, as shown in figure 9. The API exposed to untrusted code provides access to the seal method and a sealed secret function. By running ENCAP on the implementation we successfully verified that the API confines the secret function.

```
function SealerUnsealer(){
    var flag = false;
    var payload = null;
    return {seal: function (payloadToSeal){
                    function box(){
                        flag = true;
                        payload = payloadToSeal;
                    }
                    return box;
                },
            unseal: function(box){
                    flag = false;
                    payload = null;
                    try{
                       box();
                       if (!flag){
                          throw 'Invalid Box'
                       }else{ return payload;}
                    }finally{
                        flag = false;
                        payload = null;}
                }
        }
};

function secret(){ };
// a secret function

var brand = SealerUnsealer();
var box = brand.seal(secret);
// seals the secret function

var api = {seal: brand.seal, sealedFunc: box}
// API exposed to untrusted code
```

Figure 9. An Implementation of Sealer-Unsealer pairs

C. Mint

Our final example is the Mint function, which is a canonical example used in the Object-Capabilities literature to demonstrate how capability patterns like sealers and unsealers can be used for writing robust code that can be safely run in potentially malicious environments. The source code is present in figure 10. Untrusted code is handed the function Mint, which can be invoked to create the Purse constructor. The Purse constructor can be invoked to create *purse* objects which encapsulate a balance field, storing the purse's balance, and have methods deposit, getBalance to read and update the balance field. One of the correctness goals for the mint is *conservation of currency*, which says that the sum of balances of all purse objects must be constant. A quick inspection of the code reveals that the decr function can directly alter the balance field. Thus the conservation of currency property necessitates that the Mint object *safely confines* the decr function. By running ENCAP on the code in figure 10 combined with the implementation of sealer-unselars pairs from figure 9, we successfully verified that the decr function is safely confined, under the SES_{light} threat model.

D. Summary

We demonstrated the effectiveness of our tool by using it to find a security-oversight in the Yahoo! ADsafe library and

```
function Nat(n) { if (n !== n >>> 0) { throw 'NotNat'; } return n; }

function Mint(){
    var brand = SealerUnsealer();
    return function Purse(balance){
                function decr(amount){
                    balance = Nat(balance − amount);
                }
                return {
                    getBalance: function(){return balance;},
                    makePurse: function(){return Purse(0);},
                        getDecr: function(){return brand.seal(decr);},
                        deposit: function(amount,src){
                                    var box = src.getDecr();
                                    var decr = brand.unseal(box);
                                    Nat(balance + amount);
                                    decr(Nat(amount));
                                    balance += amount;}
                    }
            }
}

var api = Mint;
// API exposed to untrusted code
```

Figure 10. An Implementation of the Mint

then verifying confinement of the repaired library and some benchmark examples from the Object-Capabilities literature.

The vulnerability that ENCAP found in the ADsafe library is not only exploitable using untrusted SES$_{light}$ code, but also using code that satisfies the stronger JSLint syntactic restrictions imposed by ADsafe. In addition, the vulnerability is also exploitable on present day browsers. In the accompanying tech report [35], we use the exploit code to construct a JSLint-satisfactory script element, which when run in conjunction with the (broken) ADsafe library, is able to obtain a reference to the document object. The exploit has been tested on browsers Firefox, Chrome and Safari. Perhaps surprisingly, there exist examples of API confinement that are secure under standard JavaScript semantics but not under SES$_{light}$ semantics. For example, the following API fails to confine the function critical under the SES$_{light}$ semantics and threat model:

```
var x = function critical(){};
var api = function(){var a = this;
            if(typeof a === "object"){ delete a.x;};
            return x;}
```

However, this is safe under JavaScript semantics, for restricted untrusted code that only accesses the global variable api. This is because in the JavaScript semantics, the this value of the api function would be the global object and therefore the priv binding would get deleted before the return step. However under the SES$_{light}$ semantics, the this value would be undefined thereby making the function return critical.

Finally, we note that ENCAP has the expected limitations and imprecision associated with flow insensitive and context insensitive analysis. For instance, running ENCAP on the Cajita run-time library of the Google Caja framework [6], generated a large number of false positives as a function freeze was being called on the return variables of all the library methods. Due to context insensitivity, the return value from all calls to freeze propagated to all call sites, thereby creating too many spurious points-to edges.

VII. Related Work

There is a long history of using static analysis and language-based techniques to secure extensible software, including such notable work as Typed Assembly Language [31], Proof-Carrying Code and Software-based Fault Isolation [38]. However, this line of research has focused on providing strong guarantees about untrusted extensions, and their access to trusted interfaces to security-relevant services. Less considered have been the effects of giving an arbitrary, untrusted extension unfettered access to such trusted interfaces. Until recently, most work that considered such "API security" had centered around cryptographic security modules, and their interfaces [4]. For those cryptographic APIs, keys take the role of security-critical objects, and static analysis has been used to establish whether (or not) those keys are properly confined within the security module. This line of work has strong connections to formalisms such as BAN logic [5], where similar abstract analysis can be used to reason about all possible interactions in security protocols. As security-relevant services that expose rich interfaces are increasingly written in high-level, type-safe languages, such abstract analysis of the security properties of APIs has increasingly wider applicability.

For server-side Web software written in languages other than JavaScript, several efforts have employed static analysis for security, in particular to identify and prevent Cross-Site Scripting (CSS) attacks or SQL injection. Examples include the taint-based CSS analysis in Pixy [16], the SQL injection analysis by Xie and Aiken [41], both in the context of PHP. In addition, in the context of Java, Livshits and Lam implemented a Datalog-based analysis to establish security properties such as proper sanitization [21]. Compared to this work, JavaScript raises unique challenges, in particular due to its highly-dynamic nature. In previous work [23, 25], Maffeis et al. have analyzed various subsets of JavaScript, and defined sandboxes based on filtering, rewriting and wrapping for restricting untrusted code written in them.

In previous work by Maffeis et al. [22], a small-step operational semantics has been presented for JavaScript, based on the 3^{rd} edition of the ECMA262 standard. As mentioned earlier, our semantics of SES$_{light}$ is very similar in structure to this semantics with the main technical difference being in the modeling of scope objects. An alternate approach to defining semantics of Javascript is that of Guha et al. [12], who describe the semantics by defining a de-sugaring of the surface language to a core calculus *LambdaJS* and then providing execution rules for expressions in LambdaJS.

Recently, flow-insensitive static analysis of JavaScript code has been considered in the research efforts Staged In-

formation Flow [8] and Gatekeeper [11]. Both efforts make use of mostly-static techniques, supported by some runtime checks; in particular, Staged Information Flow leaves to runtime checks the analysis of all dynamic code and *eval*. Gatekeeper has perhaps the most similar goals to our work: it aims to constrain potentially-obfuscated, malicious JavaScript widgets that execute within a host Web page, and invoke the APIs of that Web page. Gatekeeper analysis also makes use of Datalog, in much the same way as we do in our work. Gatekeeper, however, does not statically analyze eval and does not provide a rigorous proof of soundness for their analysis. As a final point of comparison, the VEX system uses static information-flow analysis to find security vulnerabilities in Web browser extensions. Much like in the current work, VEX analysis is based on a formal semantics for a fragment of JavaScript, based on [12, 22]. Despite several similarities, VEX is fundamentally different from the current work in both its application domain, and in its technical details. VEX aims to prevent script injection attacks, and analyzes only certain types of explicit flows from untrusted sources to executable sinks; in comparison, we consider the confinement of security-critical objects. VEX static analysis is path-sensitive, context-sensitive and makes use of precise summaries. but is fundamentally unsound. In comparison, our static analysis is simpler, applies to the core of an important new JavaScript variant, and guarantees soundness.

VIII. Conclusion and Future Work

While JavaScript was originally designed for adding small scripting functions to Web pages, the Web has become dramatically more sophisticated over the past 15 years. As larger and more complex applications have become commonplace, Web application developers and users have become increasingly interested in robustness, reliability, and security of large JavaScript code bases. In this paper, we therefore study a restricted sublanguage SES$_{light}$, based on recently standardized ES5S, that we believe allows concerned programmers to develop secure applications that provide restricted access to untrusted code. In effect, we believe that SES$_{light}$ provides better support for the principle of least privilege than previous *ad hoc* subsets of JavaScript because a programmer can confine access to selected resources to a specific interface (or API).

We demonstrate the way that SES$_{light}$ supports confinement by developing precise semantics for SES$_{light}$, presenting an automated tool ENCAP that provably verifies confinement, and using ENCAP to analyze code previously defined to provide confinement in restricted forms of JavaScript. In these case studies, we found a previously undetected confinement oversight in the Yahoo! ADsafe library [9], proved confinement of a repaired version of ADsafe automatically, and demonstrated confinement for other isolation examples from the object-capability and security literature. While SES$_{light}$ requires programmers of security-critical code to use a more limited form of JavaScript, we believe the clean semantic properties of SES$_{light}$ and the power of ENCAP and other analysis methods enabled by SES$_{light}$ provide ample motivation for concerned programmers to adopt this language. In fact, the success of our tool on some existing code suggests that careful programmers may already respect some of the semantically motivated limitations of SES$_{light}$.

While our success with ENCAP demonstrates some of the advantages of SES$_{light}$, additional effort may be needed to drive interest in SES$_{light}$, In addition, further technical work can provide additional and more powerful analysis methods for versions of JavaScript that support traditional programming language properties such as the static contour model of scope and the ability to rename bound variables without changing program semantics (both of which fail for arbitrary JavaScript). For example, additional analysis methods such as object-sensitive analysis [28] and CFA2 techniques [37] may lead to more powerful tools that will aid future programmers in developing security-critical code, and other methods may allow us to provide more useful diagnostics when confinement cannot be established. We also believe that further work may allow us to extend the present tool and proofs to broader classes of untrusted code.

Acknowledgment

We thank the Google Caja team for invaluable comments and discussions. We are indebted to Shriram Krishnamurthi and anonymous reviewers for their comments and suggestions. Mitchell and Taly acknowledge the support of the National Science Foundation, the Air Force Office of Scientific Research, the Office of Naval Research, and Google, Inc.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] I. Atsushi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[3] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *Proc. of PLDI*, pages 103 – 114, 2003.

[4] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proc. of CCS*, pages 260–269, 2010.

[5] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8, 1990.

[6] Google Caja Team. Google-Caja: A source-to-source translator for securing JavaScript-based Web content. http://code.google.com/p/google-caja/.

[7] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1:146 – 166, 1989.

[8] R. Chugh, J.A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proc. of PLDI*, 2009.

[9] D. Crockford. ADsafe: Making JavaScript safe for advertising. http://www.adsafe.org/, 2008.

[10] ECMA. *ECMA-262: ECMAScript Language Specification*. Fifth edition, December 2009.

[11] S. Guarnieri and B. V. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proc. of USENIX security symposium*, pages 50–62, 2009.

[12] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *Proc. of ECOOP*, pages 126–150, 2010.

[13] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. Accepted at ESOP, 2011.

[14] D. Van Horn and H. G. Mairson. Deciding kCFA is complete for EXPTIME. In *Proc. of ICFP*, pages 275–282, 2008.

[15] D. Jang and K. Choe. Points-to analysis for JavaScript. In *Proc. of ACSAC*, pages 1930–1937, 2009.

[16] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities (short paper). In *Proc. of the 2006 IEEE S&P*, pages 258–263, 2006.

[17] J. H. Morris Jr. Protection in programming languages. *Commun. ACM*, 16:15–21, 1973.

[18] S. Krishnamurthi. Confining the ghost in the machine: Using types to secure JavaScript sandboxing. In *Proc. of APLWACA*, 2010.

[19] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16:613–615, 1973.

[20] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[21] B. V. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proc. of USENIX security symposium*, pages 1–18, 2005.

[22] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS*, pages 307–325, 2008.

[23] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *Proc. of ESORICS*, pages 505–522, 2009.

[24] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted Web applications. In *Proc. of IEEE S&P*, pages 125–140, 2010.

[25] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF*, pages 77–91, 2009.

[26] J. Midtgaard and T. Jensen. A calculational approach to control-flow analysis by abstract interpretation. In *Proc. of SAS*, pages 347–362, 2008.

[27] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *Proc. of PLDI*, pages 305–315, 2010.

[28] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[29] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[30] M. S. Miller, C. Morningstar, and B. Frantz. Capability-based financial instruments. In *Proc. of FC*, FC '00, pages 349–378, 2001.

[31] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. of POPL*, pages 85–97, 1998.

[32] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[33] J. A. Rees. A security kernel based on the lambda-calculus. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

[34] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. An operational semantics for SES_{light}. http://theory.stanford.edu/~ataly/Semantics/seslSemantics.txt.

[35] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. Technical Report http://theory.stanford.edu/~ataly/Papers/sp11TechReport.pdf, 2011.

[36] The Facebook Team. FBJS. http://wiki.developers.facebook.com/index.php/FBJS.

[37] D. Vardoulakis and O. Shivers. CFA2: A context-free approach to control-flow analysis. In *Proc. of ESOP*, pages 570–589, 2010.

[38] R. Wahbe, S. Lucco, T. E. Anderson, and S.L. Graham. Efficient software-based fault isolation. In *Proc. of SOSP*, pages 203–216, 1994.

[39] J. Whaley. BDDBDDB: Bdd based deductive database. http://bddbddb.sourceforge.net/, 2004.

[40] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of PLDI*, pages 131–144, 2004.

[41] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. of USENIX security symposium*, page 179192, 2006.