JuS: Squeezing the Sense out of JavaScript Programs

Philippa Gardner
Imperial College London
pg@doc.ic.ac.uk

Daiva Naudžiūnienė
Imperial College London
dn911@doc.ic.ac.uk

Gareth Smith
Imperial College London
gds@doc.ic.ac.uk

ABSTRACT

We introduce JuS (JavaScript under Scrutiny), a first step towards a static-analysis tool for JavaScript based on a program logic in the style of separation logic. In particular, we focus on reasoning automatically about the JavaScript variable store. Because of prototype-based inheritance and `with` statements, even reasoning about variables is not trivial in JavaScript. We evaluate our tool on examples from the Firefox test suite which illustrate the scoping mechanism of the JavaScript variable store.

Categories and Subject Descriptors

F.3.1 [**Theory of Computation**]: LOGICS AND MEANINGS OF PROGRAMS Specifying and Verifying and Reasoning about Programs; D.2.4 [**Software**]: SOFTWARE ENGINEERING Software/Program Verification

General Terms

Theory, Verification

Keywords

JavaScript, separation-logic, verification, tools

1. INTRODUCTION

JavaScript has become the most widely-used language for client-side web programming. The dynamic nature of the language makes current web code notoriously difficult to write and use, leading to buggy programs and a lack of adequate static-analysis tools. We believe that logical reasoning has much to offer JavaScript. In [12], Gardner, Maffeis and Smith introduced a program logic for reasoning about a subset of JavaScript, including challenging features such as prototype inheritance and `with`. Here, we make the first step towards automation: reasoning about JavaScript's *emulated variable store*.

Program Logic [12] We use a program logic based on separation logic. Separation logic has proven to be invaluable for reasoning about programs which directly manipulate the heap, such as C and Java programs [4, 21, 5, 10]. A key characteristic of JavaScript is that the entire state of the language resides in the object heap, in a structure that imperfectly emulates the variable store of many other programming languages. It is therefore natural to investigate the use of separation logic to verify JavaScript programs.

In order to present an accurate account of JavaScript's emulated variable store, Gardner, Maffeis and Smith produced a new fundamental adaptation of separation logic. The basic reasoning rules follow the semantics of JavaScript extremely closely. As a result, the basic reasoning rules are complex and difficult to use. To overcome this complexity natural layers of abstraction are established on top of the basic reasoning which allow us to avoid many of the hairy corner cases of the language. When reasoning about principled programs we can use these abstractions exclusively. When reasoning about arbitrary programs we can break open the abstractions and reason at a lower level.

Why do we care about the complicated JavaScript variable store? Prototype-based inheritance and `with` statements complicate the structure of the JavaScript emulated variable store. If there were no `with` statement, our lives would be much easier. In fact, the ECMAScript standard is moving that direction by forbidding `with` statements in ECMAScript 5 strict mode. So why do we still wish to reason about them? There are three main reasons. *First*, we currently work with a common subset of ECMAScript 3 and ECMAScript 5. We are in the process of porting our work to full ECMAScript 5, taking browser idiosyncrasies into account (See [6] for details). Although the main browsers currently support ECMAScript 5 strict mode, the majority of code being written today is non-strict, and follows programming patterns that were developed in the days of ECMAScript 3. *Second*, even if there is a wide acceptance of ECMAScript 5 strict mode, it is inevitable that ECMAScript 5 libraries will have to interface properly with non-strict code. We therefore believe that there is a growing need for general-purpose, expressive analysis tools for both strict and non-strict code. *Third*, power users sometimes need `with`. A good example is the Google Caja sandboxing system, which makes extensive use of ECMAScript 5 strict mode to provide the security properties required of the sandbox. The sandboxed code must be written in a variant of strict ECMAScript 5 called SES [19]. However, in order to provide the required security properties the sandboxing mechanism itself is forced to make use of `with` to sandbox the SES code.

To appear in the Second Annual Workshop on Tools for JavaScript Analysis 1 July 2013, Montpellier, France Co-located with ECOOP, ECMFA and ECSA

JuS: a Symbolic Execution Tool We introduce JuS, a symbolic execution tool for JavaScript, consisting of a core symbolic execution engine, an entailment engine and a collection of "analysis strategies". The symbolic execution engine follows in the footsteps of other separation logic tools such as Smallfoot [4], jStar [10], Space Invader [21] and Abductor [8]. Symbolic execution is a standard technique which simulates the concrete execution of a program on a *set* of concrete states by tracking logical formulae (*symbolic states*), rather than concrete values. During symbolic execution, we must often check entailment between formulae. We delegate much of our entailment checking to the separation logic theorem prover coreStar [7].

A useful feature of our tool is its collection of *analysis strategies*, which are split into *abstraction levels*. We currently provide strategies associated with three abstraction levels: the basic reasoning, the key store abstraction presented in [12], and a new StoreLet abstraction which we focus on in this paper. It should be a simple matter to add new abstractions and their strategies. Each abstraction level offers a naive strategy and a bi-abduction strategy. The naive strategy requires that the verifier provides assertion annotations to the program. The bi-abduction strategy allows for automatic inference of assertions using the bi-abduction technique introduced in [8]. Using the bi-abduction strategy, user-supplied program specifications can be incomplete or even empty. JuS discovers and adds the missing parts of the specifications.

JuS focuses on accurate reasoning about the JavaScript emulated variable store. It handles prototype-based inheritance, `with` and simple functions. Our treatment of functions is enough to fully expose the complexities of the variable store.

Case Study We evaluate our tool on examples from the Firefox test suite which illustrate the scoping mechanism of the JavaScript variable store. We demonstrate that JuS can help to identify a state where most of these tests behave differently in the presence of prototype poisoning compared to naive starting states. Using abstractions to describe the emulated variable store and incorporating the bi-abduction technique, we show that our tool can infer general specifications for the examples. Such specifications state that, if the input state satisfies the pre-condition and the program terminates, then the resulting state will satisfy the post-condition. This alleviates the need for many mundane unit tests, while interactively investigating the shape of the pre-condition suggests corner cases that could (and should!) be profitably tested.

2. THE JAVASCRIPT VARIABLE STORE

When reasoning about other programming languages, the behaviour of program variables is well understood and relatively simple. Program variables are usually modelled using a "variable stack". This stack is distinct from the program heap which contains dynamically allocated objects. In JavaScript there is no such clear distinction. What, in other languages, would be the global-most "stack frame" is an object in JavaScript. It is called the global object, and we are free to manipulate pointers to it just as we would to any other object. In addition, the `with` statement allows us to insert arbitrary JavaScript objects into the JavaScript equivalent of a variable stack. For these reasons, it is simplest to think of JavaScript program variables as living in an *emulated variable store* which exists entirely on the object heap. Instead of stack frames, we have regular objects pressed into service as *scope objects*. We maintain a list of pointers to such objects, called a *scope list*. As we will see below, some scope objects are created specifically for the purpose of variable resolution and are called *activation objects*.

When we wish to access a variable, we start at the head of the scope list and search until we either: (1) find an object that contains a field whose name matches that of the variable; or (2) reach the end of the list. This would be a simple matter if it were not for the complication that JavaScript is a "prototype-based" language. JavaScript objects may inherit properties and behaviour from other *concrete objects* at runtime in much the same way that Java objects of a given class may inherit behaviours from other *classes* known at compile time. If the JavaScript object `A` inherits behaviour from the object `B`, then `B` is said to be the *prototype* of `A`, and `A` will contain a *prototype pointer* to `B`. Since `B` may itself have a prototype, we will often refer to the *prototype chain* of `A`. This is a list of prototype pointers beginning at `A`, and containing references to all the objects from which `A` may inherit behaviour. Since scope objects are objects in the heap, they can have prototype chains. When we lookup a variable, we must investigate not only the scope objects, but also all the objects in their prototype chains. If we wish to *read* a variable, we may find ourselves reading from the prototype chain of a scope object, rather than reading directly from the scope object itself. On the other hand, if we wish to *write* to a variable we should only write to actual scope objects, and never to their prototypes. Hence, a simple variable assignment in JavaScript can be an *overriding assignment* which hides rather than over-writes the old value.

To see these mechanisms in action, let us consider an example taken from the Firefox test suite given in Figure 1. This example expects the final value of the variable `actual` to be 2. Let us see why this is the case when we start the example in the initial state.

When we start a JavaScript program, the scope list contains only the global object. We denote it by l_g. The global object can have a prototype chain, which in the initial state will normally terminate in the default object prototype. The default object prototype is often referred to in JavaScript literature as "Object.prototype". We denote it by l_{op}[1]. The length of this initial prototype chain is implementation dependent, but for this example it is sufficient to consider the case where l_{op} is the prototype of l_g. Figure 1 (a) shows such an initial state. Objects are illustrated as boxes, a dark box means that an object is currently in the scope list, prototype pointers are solid arrows, and other pointers are dashed arrows. The most local scope object is depicted at the bottom, while the most global object is at the top. Let us go through the program step by step. On line 1 we assign the value `1` to the variable `a`. First we need to find the first scope object that contains or inherits a field `a`. Since the current scope list consists only of l_g we cannot find `a` anywhere in the store. JavaScript objects are global by default, so we create `a` as a field of the global object. Similarly, on line 2 the variable `obj` is also created as a field of the global object

[1]In JavaScript "Object.prototype" really refers to the "prototype" field (which is mutable) of the variable "Object" (which is mutable and can be shadowed). For this reason we use l_{op} to refer to the initial Object.prototype object.

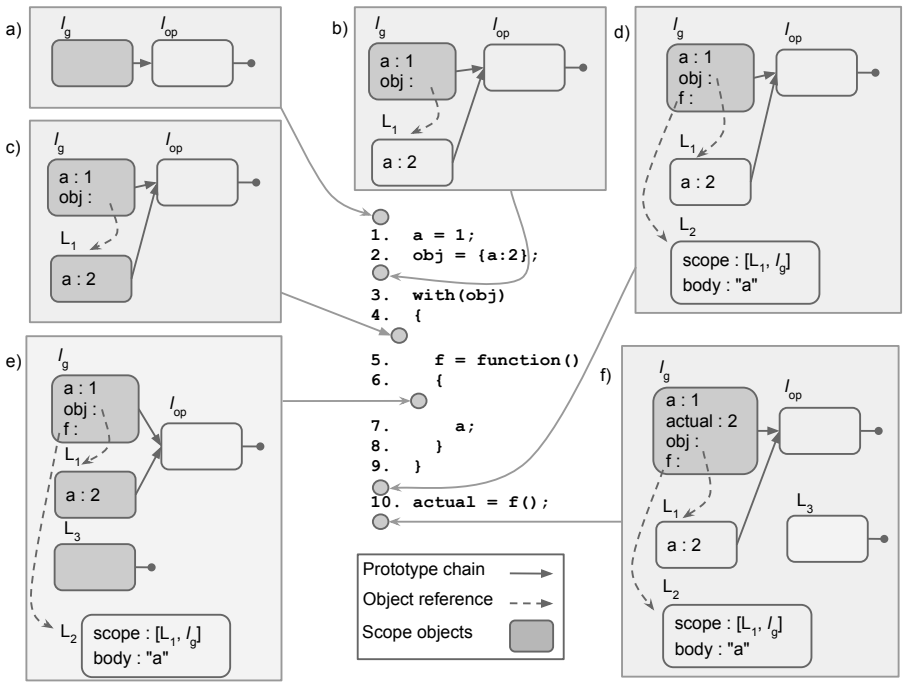

Figure 1: Example from the Firefox Test Suite.

and is assigned a newly created object that has a field `a`. All objects created using { } notation have l_{op} as their prototype. Figure 1 (b) shows the heap after these first two lines. When we execute the `with` statement, the given object is added to the beginning of the scope list (Figure 1 (c)). We execute the body of the `with` using the updated scope list $[L_1, l_g]$. On line 5 we cannot find `f` anywhere in the store, so we add it to the global object. We create a new function object which has a field `body` that contains the body of the function, and a field `scope` that saves the current scope list. This `scope` field is JavaScript's mechanism for closures. When we finish executing the body of the `with`, we restore the scope list. Line 10 is therefore executed in a scope that contains only the global object as shown in Figure 1 (d). The variable `actual` is not found anywhere in the store, so the new field will be created in the global object. The variable `f` is found as a field of the global object. It points to a function object which can be called. The function body is executed in the saved scope extended with a new object which contains any local variables or parameters the function might declare. We call this new object the *activation object* of the function. In this example the function `f` does not have any parameters or local variables so the activation object shown in Figure 1 (e) is empty. Since JuS focuses on the behaviour of the emulated variable store, we keep function control flow simple and model line 7 (the last line in a function) as an implicit return statement. We look up the variable `a` in the emulated variable store, and first find a value at location L_1. We return `2` which is assigned to variable `actual`. After the function call we restore the scope list. Figure 1 (f) shows the final heap. Activation object L_3 will be garbage collected.

Even executing such a tiny example is complicated. In the next section we describe reasoning about JavaScript using the same example.

3. REASONING ABOUT THE JAVASCRIPT VARIABLE STORE

In the last section we saw that if we execute our running example in the standard initial state, the final value of the variable `actual` is `2`. This is the expected value of the Firefox test. Does the example behave the same way no matter what variables are defined in the initial emulated variable store? What happens if we execute this example in a state that is much more complicated to start with? Reasoning can help answer these questions.

In [12] Gardner, Maffeis and Smith presented logic for reasoning about JavaScript. We present JavaScript reasoning using the symbolic execution tool JuS. We show that: (1) the tool can help in finding corner cases of the program and hence can help in providing more comprehensive test suites, (2) the tool can check and infer specifications of the programs.

3.1 Exploring Corner Cases

It is good engineering practice to write tests which cover all the corner cases of the code we write. But how are we to know where our corner cases are? Or that we have covered them all? In JavaScript, this is particularly challenging. In this section we describe how JuS may be able to help.

Let us try JuS with our example from the previous section. Figure 2 shows our example in the Eclipse IDE. We write specifications in the source file as annotations: `@toprequires` for pre-conditions and `@topensures` for post-conditions.

The first line of the pre-condition says that the current scope list (`cScope`) contains only the global object l_g[2]. The second line $\mathtt{obj}_{l_g}(|@proto : l_{op})$ says that the global object

[2]In JuS the keywords of the assertion language are prefixed with #. We omit it in the text.

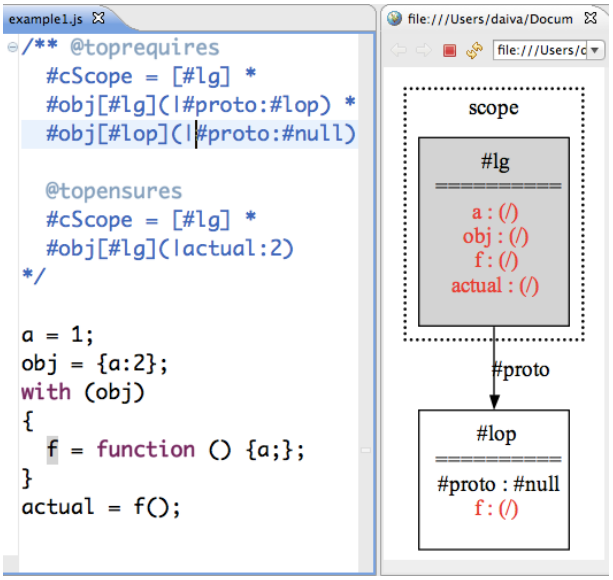

Figure 2: Example Using JuS

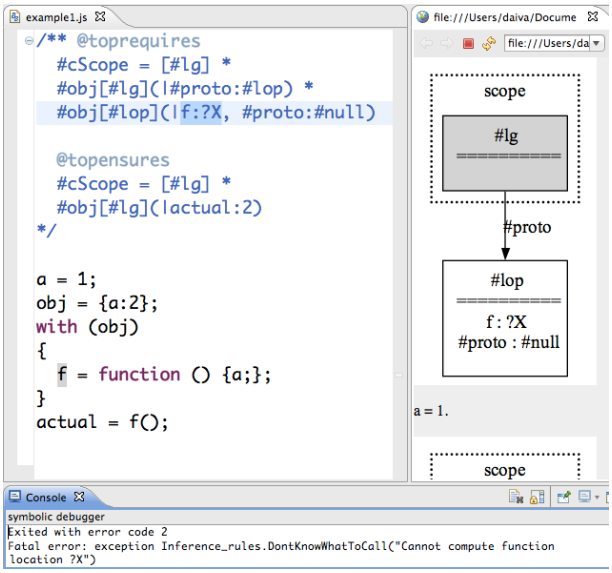

Figure 3: If Object.prototype Has Field `f`, JuS Throws an Error.

must exist, and its prototype $@proto$[3] must be l_{op} (the Object.prototype object). With `obj` notation we can explicitly say that certain fields of l_g do *not* exist, by listing them to the left of the vertical bar. For example, to say that `x` and `y` do not exist: $\texttt{obj}_{l_g}(\texttt{x},\texttt{y}|@proto : l_{op})$. As it is, the second line of the pre-condition in Figure 2 says that l_g must exist, and must have the given value of $@proto$, but may or may not have any other fields. Similarly the third line of the pre-condition says that the l_{op} object exists, and that its prototype is `null` (it terminates the prototype chain). The lines are separated by the $*$ of separation logic, which says that l_g and l_{op} are disjoint objects. This pre-condition looks similar to the initial state from Figure 1 (a). The difference is that JuS allows us to express uncertainty about the existence of most of the fields of our starting objects. The post-condition says that the field `actual` has been created in the global object and that its value is `2` as this test expects it to be. Notice that since JavaScript is a garbage collected language, JuS allows us to omit descriptions of objects which are no longer relevant to the program execution.

Our pre-condition is not enough to prove our post-condition. If we ask JuS to check this specification, it gives an error that it cannot proceed. We can either strengthen our pre-condition manually by providing additional information about the variables used in the program, or ask JuS to infer the missing parts of our specification. In this example we ask JuS to fill in the gaps. JuS generates similar diagrams to those in Figure 1. In Figure 2 we see such a diagram on the right hand side. It corresponds to the given pre-condition with additional information about the variables that was inferred by the tool. Every box corresponds to an object with its location name and its fields. We use notation `(/)` in the tool to denote an *absent* field. Grey boxes denote scope objects. The fields that were inferred by the tool are shown in red.

JuS has inferred the information it needed to proceed. To symbolically execute the program and prove its post-condition it is enough to know that the fields `a`, `obj`, `f`, `actual` do not exist in the global object and the field `f` does not exist in Object.prototype. Why should we care about the value of `f` in Object.prototype? Let us see what happens if Object.prototype has field `f` when we start the program.

Figure 3 shows our modified pre-condition where we add field `f` to the Object.prototype and say that it has some value `?X`[4]. When we symbolically execute this example, we get an error message, saying that the tool was expecting `?X` to be a location of a function it wanted to call.

Hence if we start our program in such a state, we end up calling an arbitrary function that lives in Object.prototype. This is known as "prototype poisoning" [1]. For a concrete example, consider first executing `Object.prototype.f = function () "Evil function"` and then running our example. We will end up calling the Evil `f` instead of the one that is defined in our example. Previously, when we were defining the function `f` in lines 5-8, we were in the state shown in Figure 1 (c). Since we could not find variable `f` anywhere in the store, we added it to the global object. But if we have a field `f` in l_{op} (see Figure 4 (a)), then we define `f` in the scope object L_1, since `f` is present in its prototype chain. After we leave the body of the `with` statement and call `f`, we call the function that lives in the l_{op}, since our current scope list consists only of the global object (see Figure 4 (b)).

3.2 Storelet Abstraction

In the previous section we used JuS to explore very specific pre-conditions in order to find corner cases. In this section, we introduce a way to explore very general pre-conditions. The key is to describe the JavaScript variable store in a general way that covers as many sane concrete emulated variable stores as possible.

When *reading* a variable we need to know (1) if that variable exists and (2) assuming it exists, what value does it have. When *writing* a variable we need to know enough to

[3]We use #`proto` instead of #$@proto$ in JuS.

[4]`?X` denotes universally quantified logical variable.

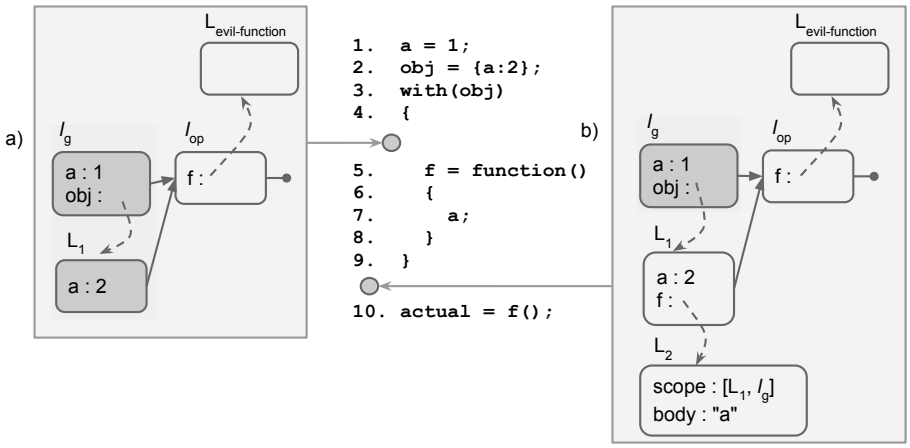

Figure 4: JavaScript Variable Store Where Object.prototype Has Field `f`

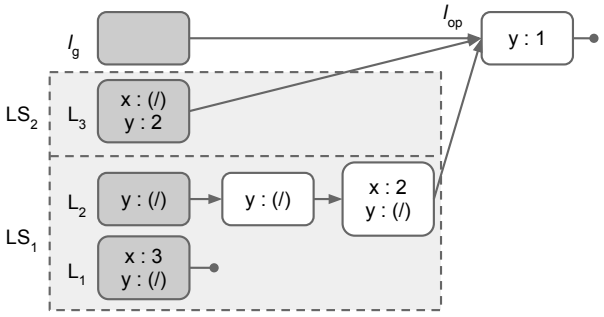

Figure 5: $\mathsf{StoreLet}_{(\mathrm{LS}_1,\mathrm{LS}_2),l_{op}}(\mathtt{y}|\mathtt{x}:3)(\mathtt{x}|\mathtt{y}:2)$

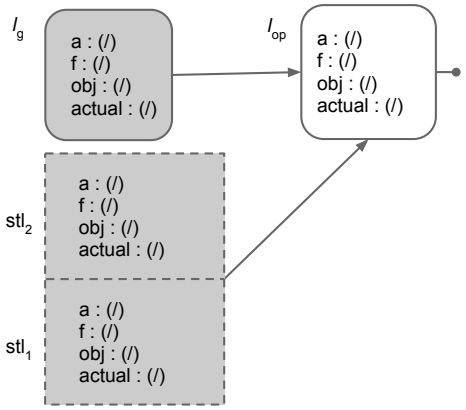

Figure 6: General Pre-condition

be able to read it again in future. In addition, recall that undeclared variables are automatically created in l_g and that l_{op} is essential for guarding against prototype poisoning. We present the StoreLet abstraction which gives us the information we need about the bulk of the emulated variable store, while keeping l_g and l_{op} separate so we can handle their special cases.

We explain StoreLet by example. Figure 5 shows a concrete variable store. A grey dashed box shows the boundary of the formula $\mathsf{StoreLet}_{(\mathrm{LS}_1,\mathrm{LS}_2),l_{op}}(\mathtt{y}|\mathtt{x}:3)(\mathtt{x}|\mathtt{y}:2)$. The StoreLet describes two sub-boxes. The first (lower) sub-box is associated with the scope list LS_1 (in this case equal to $[\mathrm{L}_1, \mathrm{L}_2]$) and the abstraction $(\mathtt{y}|\mathtt{x}:3)$. This means that if we try to read `x` in this sub-box we will find the value `3`. If we try to read the variable `y` in this sub-box, we won't find it. This sub-box ends precisely when a prototype chain escapes the StoreLet. In this example, we specify that the prototype chain must escape to l_{op}. Since we could not find `y` in our first sub-box we are forced to look for it in the escaping prototype chain. In this case we find `y : 1` in l_{op}, and can read the value `1` for the variable `y`. If `y` were not a member of l_{op}, we would proceed to examine the second (upper) sub-box, which is associated with the scope list LS_2 (in this case equal to $[\mathrm{L}_3]$) and the abstraction $(\mathtt{x}|\mathtt{y}:2)$. Notice that in this example we also know that `x` cannot be found in the second sub-box. This is a meaningful description of the state in the diagram, but is not helpful for reasoning about the program variable `x`, since this behaviour is dominated by the presence of `x` in the first sub-box. To see how this kind of formula can come about, consider running `y=4` in this state. We would end up with $\mathsf{StoreLet}_{(\mathrm{LS}_1,\mathrm{LS}_2),l_{op}}(|\mathtt{y}:4,\mathtt{x}:3)(\mathtt{x}|\mathtt{y}:2)$, in which the `y : 4` in the first sub-box dominates the `y : 2` in the second sub-box.

In explaining the example in Figure 5, we noted that the formula $\mathsf{StoreLet}_{(\mathrm{LS}_1,\mathrm{LS}_2),l_{op}}(\ldots)(\ldots)$ demands that the first prototype chain to escape the StoreLet should do so at the end of the first sub-box, and should escape to l_{op}. In fact, what the formula demands is that *every* prototype chain to escape the StoreLet must escape precisely to l_{op}, and that the first such escape marks the boundary between the two sub-boxes. Notice that while the concrete store in Figure 5 has the prototype of the object L_3 escaping the StoreLet, this makes no difference to variable resolution. Once we know where the first prototype escape happens, subsequent escapes are of no consequence because their behaviour is dominated by the first escape. This makes StoreLet ideal for separating the bulk of the messy emulated variable store from the objects l_g and l_{op} which we wish to handle separately.

We can give our example from the Firefox test suite (Figure 1) to JuS with no pre-condition at all and ask it to dis-

$$
\begin{array}{lcl}
L & \stackrel{\text{def}}{=} & \texttt{null} \mid \texttt{true} \mid \texttt{false} \mid n \mid m \\
E & \stackrel{\text{def}}{=} & \texttt{this} \mid x \mid L \mid \{x_1 \texttt{:} E_1 \ldots x_n \texttt{:} E_n\} \mid \texttt{function(}x\texttt{)\{}S\texttt{\}} \\
 & & \mid \texttt{new } E\texttt{(}E\texttt{)} \mid E\texttt{(}E\texttt{)} \mid E\texttt{.}x \mid E\texttt{[}E\texttt{]} \mid \texttt{delete } E \\
 & & \mid E \oplus E \mid E \texttt{ = } E \\
S & \stackrel{\text{def}}{=} & S\texttt{;}S \mid \texttt{var } x \mid \texttt{var } x \texttt{ = } E \mid E \mid \texttt{if(}E\texttt{)\{}S\texttt{\}else\{}S\texttt{\}} \\
 & & \mid \texttt{while(}E\texttt{)\{}S\texttt{\}} \mid \texttt{with(}E\texttt{)\{}S\texttt{\}}
\end{array}
$$

Figure 7: JavaScript subset. L, E, and S define syntax for literals, expressions, and statements. n, m, x range over numerical literals, string literals, and identifier references. Binary operator $\oplus$ can be any of $\{\texttt{+}, \texttt{-}, \texttt{===}, \texttt{<}, \texttt{<=}\}$.

cover as general pre-condition as it can. It returns the pre-condition shown in Figure 6. This pre-condition describes any variable store that does not contain the variables mentioned in the program. JuS is able to prove that, given this pre-condition, the value of the variable `actual` will be 2 after executing the program.

4. JUS: SYMBOLIC EXECUTION TOOL

JuS can be used within the Eclipse IDE, online[5] or as a command line tool. The user provides a JavaScript program and, optionally, a specification or partial specification to JuS. JuS then returns a complete specification along with diagrams if a specification could be found, or a proof failure otherwise. Figure 7 shows the JavaScript subset supported by JuS. We focus on JavaScript features that represent variable resolution. Prototype-based inheritance, `with`, and simple functions are enough to fully expose the complexity of the emulated variable store. For the specification language we use a variant of separation logic introduced by Gardner, Maffeis and Smith in [12].

The technical core of JuS is a *symbolic execution* engine, which follows in the footsteps of other separation logic tools such as Smallfoot [4], jStar [10], Space Invader [21] and Abductor [8]. Symbolic execution is a standard technique which simulates the concrete execution of a program on a *set* of concrete states by tracking logical formulae (*symbolic states*), rather than concrete values. Some symbolic execution rules, e.g. the rule for the while loop, require checking entailment between formulae. We delegate much of our entailment checking to the separation logic theorem prover coreStar [7]. After symbolically executing the code, the last step is to check that the resulting symbolic state entails any given post-condition.

The key theoretical contributions made by JuS are the StoreLet abstraction, and associated symbolic execution and bi-abduction rules. Reasoning about the grizzly details of JavaScript is extremely difficult to do by hand, let alone to automate. Most of the time we wish to reason at a higher level, so long as we know it is safe to do so. We use the StoreLet abstraction to provide this high-level reasoning, which is sound even in the presence of other code which does grizzly things with the low-level emulated variable store.

In the following sub-sections we provide some technical details about how we automated reasoning with the StoreLet abstraction.

[5]at `http://www.resourcereasoning.com/jstool.html`

4.1 Technical Details

Symbolic Execution We implement symbolic execution engine using a subset of the logic formulae from [12] as our symbolic state, and the corresponding program logic rules as our symbolic semantics. For example, the global assignment rule that is used to symbolically execute the first line in our example (Figure 1) is given in [12] as:

$$
\frac{\begin{array}{l}\text{(Assign Global)} \\ \{P\}\texttt{e1}\{R * \mathbf{r} \doteq \texttt{null}\cdot\mathrm{X}\} \\ \{R\}\texttt{e2}\{Q * (l_g, \mathrm{X}) \mapsto \varnothing * \mathbf{r} \doteq \mathrm{V}_1\} \qquad Q = S * \gamma(\mathrm{LS}, \mathrm{V}_1, \mathrm{V}_2)\end{array}}{\{P\}\texttt{e1 = e2}\{Q * (l_g, \mathrm{X}) \mapsto \mathrm{V}_2 * \mathbf{r} \doteq \mathrm{V}_2\}}
$$

We explain this rule by describing how JuS implements it. First we recursively symbolically execute `e1`. This will likely involve traversing the emulated variable store in search of some variable. If the return value in the resulting symbolic state is of the form $\texttt{null}\cdot\mathrm{X}$, it means the variable we were searching for was X, and we did not find it[6]. In general, the *reference* notation $\mathrm{L}\cdot\mathrm{X}$ denotes the field X of the object L. In this case the `null` location means we could not find the variable we were looking for, so we should create one in l_g. Next we symbolically execute `e2` to discover what value to write to our new variable. We must ensure the result of `e2` contains the required resources in its symbolic state. The key to this is the γ predicate, which we describe in the next paragraph. Finally, we update the symbolic state as specified by the rule, writing our value V_2 over the empty space $\varnothing$ that used to occupy the new field of l_g. If in this process we find that some required resource is not present, the procedure is stuck. The user must either provide more hints or activate bi-abduction (also explained below).

Reasoning About Store Traversal The program logic developed by Gardner, Maffeis and Smith [12] makes use of inductive predicates σ, π, and γ to describe the heap traversals which are central to JavaScript's variable handling. In JuS, we model these traversals using symbolic execution procedures σ_P, π_P, and γ_P. These correspond to the three distinct traversals that may be involved in a variable access. The σ_P procedure traverses the scope list and returns the first scope object to either contain or inherit a given field. The π_P procedure is used by the σ_P procedure to traverse prototype chains. It returns the first object in a given prototype chain to contain a given field. If a call to $\sigma_P(\mathrm{LS}, \mathbf{x})$ returns some location L, we can use it to construct a reference $\mathrm{L}\cdot\mathbf{x}$. For *writing* to variables, this is all we need, since variable writes may be over-riding assignments as discussed in Section 2. However, if we wish to *read* a given variable, we must traverse the prototype chain again to find its actual value. This is performed by γ_P. These procedures closely follow the structure of the corresponding predicates σ, π and γ in [12], and the semantics given in [14]. If we wish to know if some symbolic state contains the resources described by $\gamma(\ldots)$, we execute $\gamma_P(\ldots)$ in that symbolic state. If γ_P returns normally, then we know the resource is present.

As we introduce the StoreLet abstraction, we allow the scope list to contain not only object locations, but also abstract StoreLet *identifiers* $(\mathrm{stl}_1, \mathrm{stl}_2)$. In the logic, these StoreLet identifiers correspond to the two partial scope lists that the StoreLet covers. Each identifier can therefore be thought of as addressing a sub-box of a storelet. The StoreLet

[6]if the return value was otherwise, we jump to another case

$$
\begin{array}{rcll}
\sigma_P(\mathrm{stl}_1 : \mathrm{LS}, \mathtt{x}) & \triangleq & \sigma_P(\mathrm{LS}, \mathtt{x}) & \mathtt{if}\ \pi_P(\mathrm{stl}_1, \mathtt{x}) = \mathtt{null} \\
\sigma_P(\mathrm{stl}_1 : \mathrm{LS}, \mathtt{x}) & \triangleq & \mathrm{stl}_1 & \mathtt{if}\ \pi_P(\mathrm{stl}_1, \mathtt{x}) \neq \mathtt{null} \\
\\
\pi_P(\mathrm{stl}_1, \mathtt{x}) & \triangleq & \mathrm{stl}_1 & \mathtt{if}\ (\mathtt{x}, \mathtt{v}) \in \overline{\mathtt{y}} : \overline{\mathtt{v}} \\
\pi_P(\mathrm{stl}_1, \mathtt{x}) & \triangleq & \pi_P(\mathrm{L}_t, \mathtt{x}) & \mathtt{if}\ \mathtt{x} \in \overline{\mathtt{x}} \\
\pi_P(\mathrm{stl}_1, \mathtt{x}) & \triangleq & \mathrm{stl}_2 & \mathtt{if}\ \mathtt{x} \in \overline{\mathtt{x}} \wedge \pi_P(\mathrm{L}_t, \mathtt{x}) = \mathtt{null} \wedge (\mathtt{x}, \mathtt{u}) \in \overline{\mathtt{w}} : \overline{\mathtt{u}} \\
\pi_P(\mathrm{stl}_1, \mathtt{x}) & \triangleq & \mathtt{null} & \mathtt{if}\ \mathtt{x} \in \overline{\mathtt{x}} \wedge \pi_P(\mathrm{L}_t, \mathtt{x}) = \mathtt{null} \wedge (\mathrm{stl}_2 = \mathtt{null} \vee \mathtt{x} \in \overline{\mathtt{z}}) \\
\\
\gamma_P(\mathrm{stl}_1 \cdot \mathtt{x}) & \triangleq & \mathtt{undefined} & \mathtt{if}\ \pi_P(\mathrm{stl}_1, \mathtt{x}) = \mathtt{null} \\
\gamma_P(\mathrm{stl}_1 \cdot \mathtt{x}) & \triangleq & \mathtt{v} & \mathtt{if}\ \pi_P(\mathrm{stl}_1, \mathtt{x}) = \mathrm{stl}_1 \wedge (\mathtt{x}, \mathtt{v}) \in \overline{\mathtt{y}} : \overline{\mathtt{v}} \\
\gamma_P(\mathrm{stl}_1 \cdot \mathtt{x}) & \triangleq & \mathtt{v} & \mathtt{if}\ \pi_P(\mathrm{stl}_1, \mathtt{x}) = \mathrm{stl}_2 \wedge (\mathtt{x}, \mathtt{v}) \in \overline{\mathtt{w}} : \overline{\mathtt{u}} \\
\gamma_P(\mathrm{stl}_1 \cdot \mathtt{x}) & \triangleq & \gamma_P(\mathrm{L} \cdot \mathtt{x}) & \mathtt{if}\ \pi_P(\mathrm{stl}_1, \mathtt{x}) = \mathrm{L} \wedge \mathrm{L} \notin \{\mathtt{null}, \mathrm{stl}_1, \mathrm{stl}_2\}
\end{array}
$$

Figure 8: Definitions of σ_P, π_P, and γ_P for StoreLet where $P = P' * \mathsf{StoreLet}_{(\mathrm{stl}_1, \mathrm{stl}_2), \mathrm{L}_t}(\overline{\mathtt{x}} \mid \overline{\mathtt{y}} : \overline{\mathtt{v}})(\overline{z} \mid \overline{\mathtt{w}} : \overline{\mathtt{u}})$

predicate is designed to make it easy to extend σ_P, π_P, and γ_P to cover it. Recall the informal description of the semantics of JavaScript variables and StoreLets in Section 3.2. First we check the first sub-box, then the escaping prototypes, then the second sub-box. JuS maintains the invariant that whenever StoreLet identifiers appear in a scope list, they must appear in adjacent pairs and in the right order. The σ_P, π_P, and γ_P traversals are therefore easily recursively defined. We give these extended procedure definitions in Figure 8. Recall a key difference between our symbolic and concrete states: in a given concrete state, a particular field is either present in a particular object or it is not; in a given symbolic state there is a third possibility - we may not know whether a particular field is present. Notice at every step in the π_P recursion, we check whether we *know* that a given field is present, or not present in a given object or StoreLet sub-box. If it is present, we have found it. If it is not present, we continue the search. If we cannot decide, we are stuck, and must appeal to the user to either provide more information or enable bi-abduction.

Bi-abduction JuS uses the bi-abduction technique to *infer* specifications of programs. The key concept, first introduced in [8] is that of *requesting additional resource*. Suppose we are part way through symbolically executing the program `e1 ; e2` and have discovered a specification $\{P\}\mathtt{e1}\{R * F\}$. We then continue our symbolic execution by analysing `e2`. The rules for symbolically executing `e2` may require additional resources to those given by R and F. Suppose we can prove that $\{R * AF\}\mathtt{e2}\{Q\}$ holds. In this case we must request the additional resource AF from our environment. If the environment can provide that resource *separately* from the resource required by `e1` at the beginning of the program, then we know `e1` will not alter it. We call AF the "anti-frame" and we call F the "frame" of `e2`. Our final specification is $\{P * AF\}\mathtt{e1\ ;\ e2}\{Q * F\}$.

A key contribution of JuS is extending this bi-abduction procedure to handle the JavaScript emulated variable store. We augment the procedures σ_P, π_P, and γ_P with the ability to request additional resource in the event that they get stuck. The interesting case is that of π_P, which performs the leg-work of both σ_P and γ_P. We give the augmented $\hat{\pi}_p$ in Figure 9. The first two lines handle the case when we are searching for field `x` in location L, and are uncertain whether the field exists in that location or not. The first line handles the sub-case where we know that the location L has a particular prototype which we could explore if we knew for certain that `x` was not present in L. We request the additional resource $(\mathrm{L}, \mathtt{x}) \mapsto \varnothing$, which corresponds to the knowledge that `x` does not exist in the object L. The second line deals with the sub-case where we have no knowledge about the prototype of L. Even if the prototype exists, we do not know where it might point and hence would have no way of continuing our search. In this case, we had better request resource corresponding to the knowledge that `x` *does* exist in L, since that way we can safely stop our prototype traversal here. The third line is triggered when we are searching a prototype chain and do not know if it continues or not. We request resource corresponding to the knowledge that it does not. The last line describes our recursive step: we proceed under the assumption that our request for additional resource was granted, and we accumulate any additional requests made down the line.

4.2 Limitations of JuS

JuS closely follows the program logic introduced in [12] in focusing on the intricacies of JavaScript emulated variable store. JuS currently does not handle features such as type conversions, exception handling (`throw`, `try`, `catch` statements), some control flow statements (`continue`, `break`, `return`, `label`, `for`, `for in`, `switch`), higher order functions and `eval`, special native JavaScript Objects, or DOM. JuS models functions as having implicit return statements as the last statement in the function. Work is already underway to extend the semantics and program logic underlying JuS to cover these features (See [6] for more information), and separation logic tools for languages such as C and Java [8, 10, 17, 18] handle many of these features for those languages. We will take advantage of this wealth of related work in future versions of JuS.

JuS would certainly benefit from further integration with SAT-solvers and other such entailment-checking technologies. For example, JuS currently has only extremely limited support for discovering and checking proofs that require arithmetic. In Section 6 we will discuss intermediate verification languages as one promising way of providing this integration. We currently require that loop invariants are annotated but infer program specifications using abduction. In future we expect to be able to use abduction to infer loop invariants, using a similar approach to Abductor [8].

5. CASE STUDY

We tested JuS using a bundle from the Firefox test suite[7].

[7] https://developer.mozilla.org/en-US/docs/SpiderMonkey/1.8.5

$$
\begin{array}{rcll}
\hat{\pi}_P(\mathrm{L}, \mathtt{x}) & \triangleq & \mathrm{abduct}[(\mathrm{L}, \mathtt{x}) \mapsto \varnothing] & \mathtt{if}\ \ (\mathrm{L}, \mathtt{x}) \mapsto \_ \in^? P \wedge \exists \mathrm{L}'.(\mathrm{L}, @proto) \mapsto \mathrm{L}' \\
\hat{\pi}_P(\mathrm{L}, \mathtt{x}) & \triangleq & \mathrm{abduct}[\exists \mathtt{v}.(\mathrm{L}, \mathtt{x}) \mapsto \mathtt{v}] & \mathtt{if}\ \ (\mathrm{L}, \mathtt{x}) \mapsto \_ \in^? P \wedge (\mathrm{L}, @proto) \mapsto \_ \in^? P \\
\hat{\pi}_P(\mathrm{L}, \mathtt{x}) & \triangleq & \mathrm{abduct}[(\mathrm{L}, @proto) \mapsto \mathtt{null}] & \mathtt{if}\ \ (\mathrm{L}, \mathtt{x}) \mapsto \varnothing \wedge (\mathrm{L}, @proto) \mapsto \_ \in^? P \\
\mathrm{abduct}[P'] & \triangleq & r, (P' * P'') & \mathtt{if}\ \ \hat{\pi}_{P*P'}(\mathrm{L}, \mathtt{x}) = r, P''
\end{array}
$$

$\hat{\pi}_P(\mathrm{L}, \mathtt{x})$ returns values L' and P', such that $\pi_{P*P'}(\mathrm{L}, \mathtt{x}) = \mathrm{L}'$.
$(\mathrm{L}, \mathtt{x}) \mapsto \_ \in^? P$ means we can infer neither $(\mathrm{L}, \mathtt{x}) \mapsto \varnothing$ nor $\exists \mathtt{v}.(\mathrm{L}, \mathtt{x}) \mapsto \mathtt{v}$ from P.

Figure 9: Extending the Definition of π_P to $\hat{\pi}_P$ to Support Bi-abduction.

Test	Loc		Verification Time (s)			Poisoning	Inference Time (s)		
	Code	Spec	Pars./ Diag.	Symb. Exec.	coreStar		Pars./ Diag.	Bi-abd.	coreStar
Section A	8	5	1.265	0.044	0.026	y	1.144	0.049	0.074
Section B	8	5	1.238	0.037	0.028	n	1.195	0.046	0.079
Section C	8	5	1.199	0.042	0.031	y	1.171	0.050	0.080
Section D	11	5	1.198	0.042	0.035	y	1.188	0.051	0.084
Section E	9	5	1.196	0.039	0.027	y	1.262	0.050	0.076
Section F	13	5	1.237	0.049	0.099	y	1.278	0.064	0.077
Sections G&H	8	6	1.122	0.028	0.073	y	1.400	0.035	0.120

Figure 10: Results of the Case Study

The purpose of this bundle is to test scope list and identifier resolution (ECMA Section 10.1.4), the `with` statement (ECMA Section 12.10) and function definition (ECMA Section 13). Two of the eight tests in the bundle (G and H) make use of introspection features that JuS does not support. These tests differed from each other only in their use of this introspection, which was used exclusively to test the expected final state of the program. For this study we omitted these introspecting lines, and directly inspected the post-condition of the test instead.

Figure 10 shows our results. First we gave very concrete specifications that reflect the initial state for each test, and checked if our symbolic execution returned the expected results. All tests passed. It took less than 1.4 seconds to check each test. Most of this time was spent in communicating with the parser (which is written in Java, while the JuS core is written in OCaml), and in drawing diagrams. The symbolic execution itself took about 0.04 seconds. The calls to the coreStar theorem prover to check entailment took about 0.05 seconds for a test.

Next, we explored each test for corner cases as illustrated in Section 3.1. Unsurprisingly, we found prototype poisoning attacks which cause all but one of the tests to fail. This does not mean that the tests were poorly written – even well written JavaScript code is often vulnerable to such attacks. Finally, we asked JuS to infer general specifications with no help. The inferred specifications were similar in spirit to those in Figure 6 and described what we need to know about our initial state to ensure that the test will not be poisoned and that the result will be the expected one. It took about 1.4 seconds to infer these specifications, including 0.05 seconds for bi-abduction and 0.08 seconds for coreStar to check entailments.

6. CONCLUSIONS AND FUTURE WORK

We have presented JuS: the first symbolic execution tool for JavaScript using separation logic. While JuS is still in very early stages of development, it has good support for the intricacies of the JavaScript emulated variable store, and can handle tricky `with` examples automatically. It supports fully automatic bi-abductive reasoning, and a much more interactive mode of program exploration described in Section 3.1. In order to make bi-abduction possible for the JavaScript emulated variable store, we have created the StoreLet abstraction, and presented an adoption of the existing bi-abduction procedure which supports StoreLet specifically, and complex layers of abstraction in general.

Our approach is based on the program logic of [12], and the semantics of [15] and [6]. Another option would have been to use the λ_{JS} semantics of Guha *et al.* [13] as was the approach of both Fournet *et al.* [11] and Chugh *et al.* [9] in their dependant types work. We believe the heap-centric semantics of [15] better suits the separation logic approach and leads to a more natural treatment of examples that expose the tricky nature of JavaScript's emulated variable store. There are also a number of more abstract models of JavaScript, which have proven useful to study selected language features [22, 2, 20], but which are not sufficiently concrete for our purpose. We wish to reason about JavaScript from a sound semantic footing – justified both by reference to the ECMAScript specification and by empirical evidence of browser behaviour – and with a clear path to reasoning about the whole language.

Another interesting approach is to use an intermediate verification language (IVL). For example, the Javanni verifier [16] translates JavaScript programs to Boogie [3] and makes use of the verification infrastructure provided by Boogie. The tricky part is to have a correct translation of complicated JavaScript features, such as variable dereferencing and higher order functions. It is not clear from their paper if they address these problems. In future work we are planning to develop an IVL which JavaScript (with all its tricky parts) could be converted to. The IVL will be designed to make JavaScript verification easier, for example, by having

π_P, σ_P, and γ_P procedures built-in into the IVL.

This is just the beginning of the scrutiny under which we intend to place JavaScript. We are actively working on support for reasoning about higher order functions, eval, and DOM integration. We aim to produce a tool which can both (1) automatically infer specifications for many well-written programs; and (2) interactively guide a programmer through the process of discovering the corner cases in, and specifications for, more complex programs. This will allow the successors of JuS to be profitably used in conjunction with test-based engineering practices, by (1) alleviating the need for many boring tests; and (2) helping direct engineers towards interesting corner cases which they may wish to explore in their test suites.

7. REFERENCES

[1] B. Adida, A. Barth, and C. Jackson. Rootkits for javascript environments. In *WOOT*, 2009.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *ECOOP*, 2005.

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*, 2006.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.

[5] J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, 2011.

[6] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudžiūnienė, A. Schmitt, and G. Smith. The JSCert project. `http://jscert.org`.

[7] M. Botinčan, D. Distefano, M. Dodds, R. Griore, D. Naudžiūnienė, and M. J. Parkinson. coreStar: The core of jstar. In *Boogie*, 2011.

[8] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

[9] R. Chugh, D. Herman, and R. Jhala. Dependent types for javascript. In *OOPSLA*, 2012.

[10] D. Distefano and M. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.

[11] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to javascript. In *POPL*, 2013.

[12] P. Gardner, S. Maffeis, and G. Smith. Towards a program logic for JavaScript. In *POPL*, 2012.

[13] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. *ECOOP*, 2010.

[14] E. International. ECMAScript language specification. stardard ECMA-262, 3rd Edition, 1999.

[15] S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS*, 2008.

[16] M. Nordio, C. Calcagno, and C. A. Furia. Javanni: a verifier for javascript. In *Fundamental Approaches to Software Engineering*. 2013.

[17] Slayer. `http://research.microsoft.com/en-us/um/cambridge/projects/slayer/`.

[18] Space invader. `http://www0.cs.ucl.ac.uk/staff/p.ohearn/Invader/Invader/Invader_Home.html`.

[19] T. G. C. Team. How does draft ses (secure ecmascript) differ from es5? `http://code.google.com/p/es-lab/wiki/SecureEcmaScript`.

[20] P. Thiemann. Towards a type system for analyzing javascript programs. In *Proc. of ESOP*, 2005.

[21] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.

[22] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *POPL*, 2007.