Isolating JavaScript with Filters, Rewriting, and Wrappers*

Sergio Maffeis[1], John C. Mitchell[2], and Ankur Taly[2]

[1] Imperial College London
[2] Stanford University

Abstract. We study methods that allow web sites to safely combine JavaScript from untrusted sources. If implemented properly, filters can prevent dangerous code from loading into the execution environment, while rewriting allows greater expressiveness by inserting run-time checks. Wrapping properties of the execution environment can prevent misuse without requiring changes to imported JavaScript.
Using a formal semantics for the ECMA 262-3 standard language, we prove security properties of a subset of JavaScript, comparable in expressiveness to Facebook FBJS, obtained by combining three isolation mechanisms. The isolation guarantees of the three mechanisms are interdependent, with rewriting and wrapper functions relying on the absence of JavaScript constructs eliminated by language filters.

1 Introduction

Web sites such as OpenSocial [18] platforms, iGoogle [10], Facebook [7], and Yahoo!'s Application Platform [28] allow users to build gadgets, which we will refer to as *applications*, that will be served to other users when they visit the site. In the general scenario represented by these sites, application developers would like to use an expressive implementation language like JavaScript, while the sites need to be sure that applications served to users do not present security threats. In the view of the hosting site and its visitors, the containing page (for example, an iGoogle page) is "trusted," while applications included in it are not; untrusted applications could try to steal cookies, navigate the page or portions of it [3], replace password boxes with controls of their own, or mount other attacks [4]. While hosting sites can use browser *iframe* isolation, iframes require structured inter-frame communication mechanisms [3,4]. Just as OS inter-process isolation is useful in some situations, while others require language-based isolation between lightweight threads in the same address space, we expect that both iframes and language-based isolation will be useful in future Web applications. While some straightforward language-based checks make intuitive sense, JavaScript [6,8] provides many subtle ways for malicious code to subvert language-based isolation methods, as demonstrated here and in previous work [17]. We therefore believe it is important to develop precise definitions and techniques that support security proofs for mechanisms used critically in popular modern Web sites.

* This is a revised and extended version of the conference paper [14] that appeared in the Proceedings of ESORICS 2009.

In this paper, we devise and analyze a combination of isolation mechanisms for a subset of ECMA 262-3 [11] JavaScript that is comparable in expressiveness to Facebook [7] FBJS [23]. Isolation from untrusted code in our subset of JavaScript is based on filtering out certain constructs (eval, Function, constructor), rewriting others (this, e1[e2]) to allow them to be used safely, and wrapping properties (*e.g.,* object and array prototype properties) of the execution environment to further limit the impact of untrusted code. Our analysis and security proofs build on a formal foundation for proving isolation properties of JavaScript programs [17], based on our operational semantics of the full ECMA-262 Standard language (3rd Edition) [11], available on the web [13] and described previously [15]. While we focus on one particular combination of filters, rewriting functions, and wrappers, our methods are applicable to variants of the specific subset we present. In particular, DOM functions such as createElement could be allowed, if suitable rewriting is used to insert checks on the string arguments passed to eval at run-time.

While Facebook FBJS uses filters, source-to-source rewriting, and wrappers, we have found several attacks on FBJS using our methods, presently and as reported in previous work [17]. These attacks allow a Facebook application to access arbitrary properties of the hosting page, violating the intent of FBJS. Each was addressed promptly by the Facebook team within hours of our reports to them. While the safe subset of JavaScript we present here is very close to current FBJS, we consider it a success that we were able to contribute to the security of Facebook through insights obtained by our semantic methods, and a success that in the end we are able to provide provable guarantees for a subset of JavaScript that is essentially similar to one used by external application developers for a hugely popular current site.

Related work on language-based methods for isolating the effects of potentially malicious web content include [21], which examines ways to inspect and cleanse dynamic HTML content, and [29], which modifies questionable JavaScript, for a more restricted fragment of JavaScript than we consider here. A short workshop paper [27] also gives an architecture for server-side code analysis and instrumentation, without exploring details or specific methods for constraining JavaScript. The Google Caja [4] project uses an approach based on transparent compilation of JavaScript code into a safe subset with libraries that emulate DOM objects. Additional related work on rewriting-based methods for controlling the execution of JavaScript include [19]. Foundational studies of limited subsets of JavaScript and dynamic languages in general are reported in [2,25,29,9,22,1,26]; see [15]. In previous work [17], we described problems with then-current FBJS and proposed a safe subset based on filtering alone. The present paper includes a new FBJS vulnerability related to rewriting and extends our previous analysis to rewriting and wrapper functions. This produces a far more expressive safe subset of JavaScript. The workshop paper [16] describes some intermediate results on rewriting without wrappers.

The rest of this paper is organized as follows. In Section 2, we describe the basic isolation problem, our threat model, and the isolation mechanisms we use. In Section 3, we briefly review our previous work [15] on JavaScript operational semantics and discuss details of JavaScript that are needed to understand isolation problems and their solution. In Section 4, we motivate and define the specific filter, rewriting, and wrapper mechanism we use and state our main theorem about the isolation properties they provide. In Section 5, we compare our methods to those used in FBJS, with discussion of related work in Section 6. Concluding remarks are in Section 7.

2 The JavaScript Isolation Problem

The isolation problem we consider in this paper arises when a hosting page P_{host} includes content $P_1, \ldots, P_k$ from untrusted origins that will execute in the same JavaScript environment as P_{host}. We assume that $P_1, \ldots, P_k$ may try to maliciously manipulate properties of objects defined or used by P_{host}, and therefore consider $P_1, \ldots, P_k$ under control of an attacker. The isolation mechanisms we provide are intended to be used by a site that has access to $P_1, \ldots, P_k$ before they loaded in the browser execution environment. In practice, this may be achieved if the page and its constituents are aggregated at a site, or if there is some proxy in front of the browser that identifies and modifies trusted and untrusted JavaScript. While Facebook is a good example, with trusted content developed by Facebook containing untrusted user-defined applications, we develop general solutions that can be used in other scenarios that allow untrusted JavaScript to be identified and processed in advance of rendering and execution of content.

The basic defenses we provide involve changing the definitions of objects or properties in the hosting page P_{host} so that untrusted components $P_1, \ldots, P_k$ run in a modified environment, filtering $P_1, \ldots, P_k$ so that they must be expressed in a restricted subset of JavaScript, or rewriting $P_1, \ldots, P_k$ to change their semantics in some way. While potentially dangerous constructs can be eliminated by filtering, allowing them to be rewritten may provide greater programming expressiveness. While generally there may be an arbitrary number of untrusted components, we will simplify notation and discuss the problem of a program P_{host} containing two untrusted subprograms P_1 and P_2. We consider two untrusted subprograms instead of one because it is important to account for possible interaction between P_1 and P_2.

2.1 Attacker model

An attacker may design malicious JavaScript code that runs in the context of a honest page. If the honest page contains two untrusted subprograms P_1 and P_2 from different origins, then these may both be under control of a single attacker, or one may be honest and the other provided by the attacker. In the event that P_1 is honest and P_2 malicious, for example, the attacker is considered successful if execution of P_2 accesses or modifies sensitive properties of either P_1 or the hosting page P_{host}.

2.2 Sensitive Properties and Challenges

In general, different hosting pages may have different security requirements, and application developers may wish to express security requirements in some way. However, expressing and enforcing custom policies is beyond the scope of this paper. Instead, we focus on protecting a hosting page and any honest components in the following ways.

Restricting Access to Native Properties. While memory safety is often the bottom line for language-based isolation mechanisms, JavaScript does not provide direct access to memory locations. The analogous bottom line for JavaScript isolation is preventing an attacker with control of one or more applications from accessing security-critical properties of native objects (in the context of web pages, this will also include DOM objects) used by the hosting page or by other applications.

In JavaScript, there are three ways to directly access a property x of a generic object o: by o.x, by o["x"], or by the identifier expression x if o is part of the current scope chain. Certain native objects such as Array, Function, and a few others can also be accessed indirectly, without naming a global variable. Although for certain purposes some of them may have to be made inaccessible, these objects themselves do not constitute sensitive resources *per se*. Therefore, we focus on direct access to native objects. In doing so, we assume that the hosting page has a list of security critical properties, which we call *blacklist* $\mathcal{B}$. Thus the first part of our isolation goal (formally stated in Section 4) is to prevent untrusted code from accessing any properties from the list $\mathcal{B}$. Although the isolation problem and the solution proposed in this paper are parametric on a blacklist, the way our solution is designed, it is completely straightforward to transform the solution to instead apply to a *whitelist* which is the set of all properties of native objects that *can* be exposed to untrusted code.

Isolating the Namespace of Untrusted Principals. In our attacker model, a malicious application succeeds in attacking the system also if it can access properties defined by other honest applications. All untrusted application code is executed in the same global scope. Therefore, a secondary isolation goal is to separate out the set of global variables accessed by any two untrusted programs coming from different origins. In the solution we propose, we assume that each untrusted program P has an id pid_P associated with it which is unique for each origin, and we prefix all identifiers appearing in the program P with pid_P. This effectively separates the namespaces of two programs with different pids.

2.3 Enforcement Techniques

We analyze and prove the correctness of three techniques that are effective in protecting sensitive properties of honest code against an attacker that supplies code to be executed in the same JavaScript environment.

Filtering. Untrusted code may be statically analyzed and rejected if it does not conform to certain criteria. In principle, filtering may range from simple syntactic checks to full-fledged static analysis, with obvious tradeoffs between efficiency and precision. Filtering takes place once, before untrusted code is loaded into the execution environment. Since filtering does not modify code, it does not affect the performance or the behavior of untrusted code that passes the filter.

Rewriting. Selected constructs within untrusted code may be re-written. Typically, rewriting inserts run-time checks that prevent undesirable actions. While run-time checks impose a performance penalty, they are a valuable option for constructs that are potentially dangerous but also useful when used appropriately in honest code. Rewritten code may execute differently from the original code, for example when a run-time security violation is detected.

Wrapping. Sensitive resources of the trusted environment can be wrapped inside functions that use run-time checks to ensure that these resources are not used maliciously by untrusted code. Wrapper functions do not alter the untrusted code. When trusted code can access the wrapped resources directly, bypassing the wrappers, the run-time overhead or other down-sides of wrapping can be limited to untrusted code.

3 Design principles

In this Section we informally summarize the key features and insights that we gained while formalizing the operational semantics of JavaScript [13,15] based on the ECMA-262 standard [11].

We denote the ECMA-262 compliant subset of JavaScript by JS_{E2}. This paper deals with subsets of JS_{E2}. Our operational semantics consists of a set of rules written in a conventional meta-notation suitable for rigorous but (currently) manual proofs. Given the space constraints, we only describe informally the semantics of some of the unusual and interesting constructs which will help us in designing the isolation enforcement mechanisms in Section 4. Note that besides all terms derivable from the grammar (called *user* terms), our semantics introduces also certain internal terms, objects and properties useful to clearly express the evaluation semantics of user terms. None of these internal terms, objects and properties are visible in user code. Throughout the semantics, we use the symbol @ to distinguish user terms from internal terms.

Notations and Conventions. Our semantics is a small-step operational semantics ([20]). We represent objects as records of values *ov* indexed by strings m or internal identifiers @x. The record indexes are also called object properties. In JavaScript everything, including functions, is represented as an object. In our semantics the memory (or *heap* H) is a mapping from heap address (l) to objects. Object values (ov) are either pure values (pv) or function descriptions fun(x,...){P} or heap addresses. We refer to the union of the set of primitive values and heap addresses by va.

We use H_0 to denote the initial heap of JS_{E2}. It contains native objects for representing predefined functions, constructors and prototypes, and the global object @Global that constitutes the initial scope, and is always the root of the scope chain. For example, the global object defines properties to store special values such as &NaN and &undefined, functions such as eval and constructors to build generic objects, functions, numbers and arrays. In browsers, the global object is called window. We use l_g to denote the heap address of the global object.

The scope and prototype chains are two distinctive features of JavaScript. The stack is represented by a chain of objects whose properties represent the binding of local variables in the scope. Each scope object stores a pointer to its enclosing scope object in an internal @Scope property. Representing the stack as a chain of scope objects helps in dealing with the semantics of constructs that modify the scope chain, such as function calls and the with expression. JavaScript follows a prototype-based approach to inheritance. In our semantics, each object stores in an internal property @Prototype a pointer to its prototype object, and inherits its properties. At the root of the prototype tree there is @Object.prototype, that has a null prototype. There are also other native prototype objects such as Function.prototype, Array.prototype etc., which are present at the top of the prototype chains for function, array objects.

We represent a program state as a triple (H, l, t) where H denotes the heap mapping locations to objects, l denotes the heap address of the *current scope object* and t denotes the term being evaluated. Terms t can be expressions, statements and programs. We use the notation $\mathcal{H}(S)$, $\mathcal{S}(S)$ and $\mathcal{T}(S)$ to denote heap, scope and term component of the state respectively.

The general form of an evaluation rule is $\frac{<Premise>}{S_1 \rightarrow S_2}$, meaning that if a certain premise is true then the state S_1 evaluates to a state S_2. The premises of each semantic rule are predicates that must hold in order for the rule to be applied, usually

built of very simple mathematical conditions such as set membership, inequality and semantic function application. The semantics of programs depends on the semantics of statements which in turn depends on the semantics of expressions which in turn, for example by evaluating a function, depends circularly on the semantics of programs. These dependencies are made explicit by contextual rules, that specify how a transition derived for a term can be used to derive a transition for a larger term including the former as a sub-term.

An atomic transition is described by an axiom. For example, the axiom

$$\mathsf{H,l,(v)} \longrightarrow \mathsf{H,l,v}$$

describes that brackets can be removed when they surround a value (as opposed to an expression, where brackets are still meaningful).

Contextual rules propagate such atomic transitions. For example, if program H,l,P evaluates to H1,l1,P1 then also H,l,@FunExe(l2,P) (an internal expression used to evaluate the body of a function) evaluates to H1,l1,@FunExe(l2,P1). The rule below shows that @FunExe(l,−) is one of the contexts eCp for evaluating programs.

$$\frac{\mathsf{H,l,P} \xrightarrow{P} \mathsf{H1,l1,P1}}{\mathsf{H,l,eCp[P]} \xrightarrow{e} \mathsf{H1,l1,eCp[P1]}}$$

The full formal semantics [13] contains several other contextual rules to account for other mutual dependencies and for all the implicit type conversions. This substantial use of contextual rules greatly simplifies the semantics and will be very useful to prove its formal properties.

A *reduction trace* τ is the (possibly infinite) maximal sequence of states $S_1, \ldots, S_n, \ldots$ such that $S_1 \rightarrow \ldots \rightarrow S_n \rightarrow \ldots$. Given a state S, we denote by $\tau(S)$ the (unique) trace originating from S and, if $\tau(S)$ is finite, we denote by $Final(S)$ the final state of $\tau(S)$.

3.1 Property access

We now describe the semantics of various constructs which involve accessing properties of objects. By "accessing a property" we refer to either reading or writing the contents of the property. The evaluation of certain constructs, such as p in o, involve checking if the object o has a property p. We do not consider those events instances of property access. Property accesses can be *explicit* or *implicit*.

Explicit property access.. These take place when a term explicitly names the property that is being read.

Fact 1 *There are only three kinds of expressions in JS_{E2} which can be used for explicit property access: x, e.x and e1[e2].*

Below, we discuss the semantics of each of the expressions x, e.x and e1[e2].

- x: This is the standard identifier expression. Its semantics is based on the scope and prototype lookup mechanism. The evaluation involves successively looking at objects on the scope chain, starting from the current scope object until we find an object which has the property x (either in it or in one of its prototypes). Thus the expression x can potentially involve access to property "x" of one of the objects (or its prototype) present on the current scope chain.

– e.x: This is the standard *dot* notation based property access mechanism. Executing this expression will result in accessing property "x" of the object obtained by evaluating the expression e.
– e1[e2]: This is the most unusual property access mechanism. It involves accessing the property name corresponding to the string form of the value obtained by evaluating e2. Thus the property that is accessed is constructed dynamically by evaluating an expression. The actual evaluation of e1[e2] goes through the following steps (informally): First e1 is evaluated to a value va1, then e2 to va2, then if va1 is not an object it is converted into an object o, and similarly if va2 is not a string it is converted into a string m. Finally property m of object o is accessed:

  $$\mathsf{e1[e2]} \longrightarrow \mathsf{va1[e2]} \longrightarrow \mathsf{va1[va2]} \longrightarrow \mathsf{o[va2]} \longrightarrow \mathsf{o[m]}$$

  Each of these steps, which precede the actual access of property m in o, may raise an exception or have other side effects.

Implicit property access.. These take place when the property accessed is *not* named explicitly by the term, but is accessed as part of an intermediate evaluation step in the semantics. For example, the toString property is accessed implicitly by evaluating the expression "a"+ o, which involves resolving the identifier o and then type converting it to a string, by calling its toString property.

There are many other expressions whose execution involves implicit property accesses to native properties, and the complete set is hard to characterize. Instead, we enumerate the set of all property names that can be implicitly accessed.

Fact 2 *[15]. The set of all property names $\mathcal{P}_{nat}$ that can be accesses implicitly by JS_{E2} constructs is {0,1,2,...} $\bigcup$ {toString, toNumber, valueOf, length, prototype, constructor, message, arguments, Object, Array, RegExp}*

3.2 Dynamic code generation

For example, the native function eval takes a string as an argument, parses it as a program, and evaluates the resulting program returning its final value. According to the operational semantics, in JavaScript there are only two constructs which can dynamically generate new code.

Fact 3 *The only JS_{E2} constructs which involve dynamic code generation (from strings to Programs) are the native functions pointed to by the properties eval and Function of the global object.*

3.3 Accessing the global object

Since controlling access to global object is crucial in isolating untrusted from trusted code, we explore the set of constructs that can be used to access the global object.

As our semantics is formulated, the global object for the initial heap state is only accessible via the internal properties @scope and @this. These internal properties can only be accessed as a side effect of the execution of other instructions. An analysis of our semantics shows that the contents of the @scope property are never returned as the final result of any evaluation step, and the only construct whose evaluation involves access to the @this property is the expression this.

Besides using this, the global object can be returned by calling in the global scope the functions valueOf of Object.prototype, and concat, sort or reverse of Array.prototype. For example, var f=Object.prototype.valueOf; f() evaluates to the global object.

Fact 4 *The only JS_{E2} constructs that can return a pointer to the global object are: the expression this, the native method valueOf of Object.prototype and native methods concat, sort and reverse of Array.prototype.*

4 A safe subset of JavaScript

In this Section, we formally state the isolation problem introduced in Section 2.2, and propose a solution based on filtering, rewriting and wrapping techniques.

As mentioned in Section 2, we consider web pages which include untrusted content $P_1, \ldots, P_k$ in the JavaScript environment of the host page. We associate to each untrusted user program P a unique identifier pid_P, which corresponds to the origin from which the program was loaded. Given a heap H, let $Access(H, P)$ be the set of property names accessed when P is executed against H in the global scope, and let $Access_l(H, P)$ ($l \in dom(H)$) be the set of properties of the object at address l, accessed when P is executed against the heap H in the global scope.

Isolation Problem *Given a blacklist $\mathcal{B}$ of property names, find a meaningful subset $J_{sub}(\mathcal{B}) \subseteq JS_{E2}$, an appropriate wellformed initial heap state ${H_0}^{sub}$ and a function $Enf : pid * J_{sub}(\mathcal{B}) \rightarrow JS_{E2}$ such that*

(Goal 1) *For all user programs P in the subset $J_{sub}(\mathcal{B})$ with program ids pid_P*

$$Access(H_0, Enf(pid_P, P)) \cap \mathcal{B} = \emptyset.$$

(Goal 2) *For any two untrusted programs P_1 and P_2 in the subset J_{sub} with program ids pid_{P_1} and pid_{P_2} respectively*

$$Access_{l_g}(H_0, Enf(pid_{P_1}, P_1)) \cap Access_{l_g}(H_0, Enf(pid_{P_2}, P_2)) \subseteq \mathcal{P}_{nat} \cup \mathcal{P}_{noRen}.$$

Goal (2), as stated above, is the most precise property isolating different applications that we are able to support using the current proof techniques. In future work, we plan to generalize this property to enforce isolation when the execution of applications is interleaved, introducing proof techniques able to handle the combination of alternative safety properties for each application.

4.1 Isolating blacklisted properties

In order to achieve Goal 1, we need to control all possible ways in which object properties can be accessed.

As discussed in Section 3, there are two kinds of property accesses: explicit and implicit access, and for isolating blacklisted properties we need to control both of them. The implicit accesses are in general very difficult to control because given a term t, it is undecidable to statically decide the precise list of property names that will be accessed implicitly. On the positive side, from Fact 2, we know that the set of property names that would be accessed implicitly would be contained in the set $\mathcal{P}_{nat}$. In this

work, we therefore assume that none of the properties from the set $\mathcal{P}_{nat}$ are blacklisted or in other words all implicit property accesses are considered safe and are allowed.

From Fact 1 we know that x, e.mp and e1[e2] are the only expressions which can be used for explicitly accessing user properties. Hence, in order to restrict access to blacklisted properties we have to restrict the behavior of these expressions.

In this work we combine the filtering approach of [17] to restrict the behavior of expressions x and e.x with a rewriting based approach to restrict the behavior of e1[e2].

Restricting x and e.x. The expressions x and e.x can access a blacklisted property if the identifier name *"x"* is contained in the blacklist. In order to restrict this behavior we conservatively disallow all such expressions where *"x"* is contained in the blacklist.

Filter 1 *Disallow all terms which contain an identifier from the blacklist $\mathcal{B}$.*

This restriction mechanism will fail if dynamically generated code can contain blacklisted identifiers. From Fact 3 we know that JS_{E2} includes two primitive functions which can be used to generate code dynamically. One approach to fixing this problem is to restrict all ways of accessing such functions. In the initial heap, this can be achieved by disallowing the identifiers eval, Function and constructor.

Although this may be a restriction for full-blown JavaScript applications that use eval to parse JSON code, a recent study by Livshits and Guarnieri[12] shows that a low percentage of widgets use constructs like eval. Thus, we propose the following filtering step.

Filter 2 *Disallow all terms containing any of the identifiers eval, Function, or contructor.*

An alternative to the above filtering step is to define safe wrappers for the functions eval and Function. Such wrappers need to use a JavaScript expression to parse, filter and rewrite the string passed as an argument to the original functions. Proving such a JavaScript expression correct would complicate severely our analysis, and we leave for future work.

Restricting e1[e2]. We restrict the behavior of e1[e2] by rewriting it to a safe expression. The main idea is to insert a run-time check in each occurrence of e1[e2] to make sure that e2 does not evaluate to a blacklisted property name. We transform every access to a blacklisted property of an object into an access to the property *"bad"* of the same object (assuming $\mathcal{B}$ does not contain *"bad"*). Although this transformation seems easy, it is complicated by subtle details of the semantics of the expression e1[e2].

In view of our operational semantics for e1[e2] we propose the following rewriting step.

Rewrite 1 *Rewrite every occurrence of e1[e2] in a term by e1[IDX(e2)], where,*

```
IDX(e2) = ($=e2,{toString:function(){return ($=$String($),CHECK_$)})
CHECK_$ = ($BL[$] ? "bad":
          ($ == "constructor" ? "bad":
          ($ == "eval" ? "bad":
          ($ == "Function" ? "bad":
          ($[0] == "$" ? "bad":$)))))
```

where $String *refers to the original* String *constructor,* $BL *is a (blacklisted) global variable containing an object with all blacklisted property names initialized to true, and* $ *is a reserved variable name.*

In order to initialize the variables $String and $BL to their appropriate values, we propose the following (trusted) initialization code, that must be executed in the global scope of the initial heap.

Initialization Code 1 (T_{idx}) *Let* {p_1,...,p_n} *be the blacklist* $\mathcal{B}$.

var $String = String; var $= ""; var $BL = {p_1:true;...,p_n:true}.

The IDX code defined in the rewrite rule work as follows: evaluates (once and for all) e2 to a value va2 that is saved in the variable $. It then creates a new object with a specially crafted toString property, and returns the address of this object as the final value l2. These steps correspond to the internal execution trace

$$\mathsf{e1}[\mathsf{IDX}(\mathsf{e2})] \longrightarrow \mathsf{va1}[\mathsf{IDX}(\mathsf{e2})] \longrightarrow \mathsf{va1}[\mathsf{l2}] \longrightarrow \mathsf{o}[\mathsf{l2}]$$

According to the JavaScript semantics, the evaluation of o[l2] involves converting the object at address l2 to a string by calling the toString method of l2 that will return the result of converting $ to a (sanitized) string. The conversion to a string is faithfully implemented by the expression $String($), which calls the native String method on $. The expression CHECK_$, uses nested conditional expressions to return the string saved in $ only if it is not set a blacklisted property.

To protect this mechanism from tampering, we also need to ensure that the properties $, $String and $BL cannot be accessed by untrusted code. Similar restrictions need to be imposed on other variables needed by similar enforced mechanisms. Therefore, we impose the restriction that untrusted code cannot use identifier names beginning with $, thus separating the namespaces of trusted and untrusted code.

Filter 3 *Disallow all terms which involve an identifier name beginning with* $.

Note that the condition $[0] == "$"? "bad":$ in the CHECK_$ expression already imposes this restriction on dynamically generated property names.

4.2 Isolating one program from another

In order to achieve Goal 2, we need to make sure that for two programs P_1 and P_2 with ids $pid_{P_1} = pid_{P_2}$, it is the case that

$$Access_{l_g}({H_0}^{sub}, Enf(pid_{P_1}, P_1)) \cap Access_{l_g}({H_0}^{sub}, Enf(pid_{P_2}, P_2)) = \emptyset$$

where $Access_{l_g}(H_0, Enf(P_1))$ refers to the set of global object properties (or global variables) that are accessed during the entire evaluation trace of program P. As discussed in the previous subsection, it is very difficult to control implicit property accesses. Therefore we assume that accessing the same properties from the set $\mathcal{P}_{nat}$ is safe for both programs and weaken our goal to the following

$$Access_{l_g}({H_0}^{sub}, Enf(pid_{P_1}, P_1)) \cap Access_{l_g}({H_0}^{sub}, Enf(pid_{P_2}, P_2)) \subseteq \mathcal{P}_{nat}.$$

On analyzing our semantics, we found that properties of the global object can be accessed in following two ways:

(1) **Explicit access to the global object.** If the program can get a pointer l_g to the global object then it can access properties of the global object directly by using one of the two expressions l_g.x or l_g[x]. We isolate the property names accessed using these expressions by conservatively disallowing explicit access to the global object by untrusted code.
(2) **Scope resolution.** Since the global object is also the base scope object, variable names appearing in a program can resolve to the global object thereby resulting in access to the corresponding property. In other words, evaluation of the expression x can potentially involve accessing the property x of the global object. We isolate the set of property names accessed in this way by uniquely prefixing all identifiers appearing in a program by its id, thereby separating out the namespaces of two programs with different ids.

From Fact 4 we know that a pointer to the global object can potentially be obtained by using the expression this or calling method valueOf of Object.prototype or methods sort,reverse,concat of Array.prototype. In [17] we used the filtering approach and conservatively disallowed this and the identifiers valueOf, sort,reverse,concat from the language. In this work, we use the rewriting technique for restricting the behavior of this and the wrapping technique for the native methods.

Rewriting this. The main idea is to rewrite every occurrence of this in the user code to the expression NOGLOBALTHIS which returns the result of evaluating this, if it is not the global object, and null otherwise.

Rewrite 2 *Rewrite every occurrence of this by NOGLOBALTHIS, where NOGLOBALTHIS = (this==$g?null;this). and $g is a blacklisted global variable, initialized with the address of the global object.*

In order to initialize correctly $g with the global object, we use the following initialization code that must be executed in the global scope.

Initialization Code 2 (T_{ng}) `var $g = this;`

Note that, Filter 3 and Rewrite 1 already enforce that untrusted code cannot access the trusted variable name $g.

Wrapping Native methods. As opposed to [17], in this work we take the less conservative approach of wrapping the native methods in order to ensure that the value returned by them is never the heap address of the global object. The following trusted initialization code demonstrates the wrapping for the method valueOf.

Initialization Code 3 ($T_{valueOf}$)

```
$OPvalueOf = Object.prototype.valueOf;
$OPvalueOf.call = Function.prototype.call;
Object.prototype.valueOf =
        function(){var $= $OPvalueOf.call(this); return ($==$g?null:$)}
```

The main idea is to redefine the method to a new function which calls the original valueOf method and returns the result only if it is not the global object. We store a pointer to the original valueOf and call methods and the global object using $-variable

names. Since untrusted code is restricted from accessing $-properties (see Filter 3 and Rewrite 1), these are automatically isolated form untrusted code. Similarly we can define the appropriate initialization code for the methods sort, concat, reverse of Array.prototype. We denote these by T_{sort}, T_{concat} and $T_{reverse}$.

Restricting identifier names. In order to make sure that the identifier names appearing in a program P are distinct from the ones occurring in another program with a different pid, we essentially rewrite all identifiers x to pid_x.

Although this will completely separate the namespaces of any two programs with different pids, thereby achieving the isolation goal, blindly renaming all identifiers will drastically modify the semantics of the program including that of good programs. The most obvious example is the expression toString(), that evaluates to *"[object Window]"* in the un-renamed version, whereas it raises a reference error exception when it is evaluated as a12345_toString() in the renamed version. The main issue is that variable names are in fact properties of the scope object or of the prototypes of the scope objects. Since the native properties of the global object and prototype objects are not renamed, the corresponding variable names in the program should also not be renamed, in order to preserve this correspondence between them. By analyzing the semantics, we found the complete set of property names that should not be renamed as, denoted by $\mathcal{P}_{noRen}$, to be

$$\left\{ \begin{array}{l} \text{NaN,Infinity,undefined,eval,parseInt,parseFloat,IsNaN,} \\ \text{IsFinite,Object,Function,Array,String,Number,Boolean,} \\ \text{Date,RegExp,Error,RangeError,ReferenceError,TypeError,} \\ \text{SyntaxError,EvalError,constructor,toString,toLocaleString,} \\ \text{valueOf,hasOwnProperty,propertyIsEnumerable,} \\ \text{isPrototypeOf} \end{array} \right\}$$

Since we do not rename the variable whose names appear in $\mathcal{P}_{noRen}$, we can only enforce the weaker isolation

$$Access_{l_g}({H_0}^{sub}, Enf(pid_{P_1}, P_1)) \cap Access_{l_g}({H_0}^{sub}, Enf(pid_{P_2}, P_2)) \subseteq \mathcal{P}_{nat} \cup \mathcal{P}_{noRen}$$

and rely on the assumption that it is safe for two untrusted programs to access the same set of non-blacklisted native properties of the global object. In particular, eval and Function are always filtered out by Filter 2.

Thus, we propose the following rewriting step.

Rewrite 3 *Given a program P, rewrite all identifiers x $\notin \mathcal{P}_{noRen}$, appearing in P to pid_Px.*

4.3 Defining $J_{sub}(\mathcal{B})$, ${H_0}^{sub}$ and Enf

We now combine the filtering, rewriting and heap initialization steps mentioned in the previous section to define the subset $J_{sub}(\mathcal{B})$, the initial heap ${H_0}^{sub}$ and the enforcement function enf, which together solve the isolation problem. By design, the steps proposed in the previous subsection are all compatible with each other and can be combined in a straightforward manner. Based on the filtering steps, we propose the following definition for the subset $J_{sub}(\mathcal{B})$.

Definition 1. *[$J_{sub}(\mathcal{B})$] Given a blacklist $\mathcal{B}$, the subset $J_{sub}(\mathcal{B})$ is defined as JS_{E2} MINUS: all terms containing identifiers from the set $\mathcal{B}$, all terms containing one or more of the identifiers {eval, Function, contructor}, all terms containing identifiers beginning with $.*

Based on the rewriting steps, we define the function Enf as follows:

Definition 2. *[Enf] Given a program P we define, $Enf(pid_P, P)$ as program P with*

(1) *Every occurrence of the expression e1[e2] is rewritten to e1[IDX(e2)],*
(2) *Every occurrence of the expression this is rewritten to NOGLOBAL(this)*
(3) *Every identifier x appearing in the program must be replaced with pid_Px if $x \notin \mathcal{P}_{noRen}$.*

Combining all the initialization steps we define the initialized heap ${H_0}^{sub}$ as:

Definition 3. *[${H_0}^{sub}$] Given the initial JS_{E2} heap H_0, we define ${H_0}^{sub}$ as the heap obtained after executing all the initialization codes in the global scope. Formally, ${H_0}^{sub} = \mathcal{H}(Final(H_0, l_g, T_{idx}; T_{ng}; T_{valueOf}; T_{sort}; T_{concat}; T_{reverse}))$.*

Note that for correctness of our solution, it is very important to execute the trusted initialization code on the initial JS_{E2} heap H_0 (described in Section 3) and hence *before* any untrusted code is executed.

Theorem 1 (Isolation theorem). *Given a blacklist $\mathcal{B}$, such that $\mathcal{B} \cap \mathcal{P}_{nat} = \emptyset$, and the subset $J_{sub}(\mathcal{B})$, function Enf and the heap ${H_0}^{sub}$ as defined in Definitions 1, 2 and 3 respectively.*

(1) *For all user programs P in the subset $J_{sub}(\mathcal{B})$ with program ids pid_P*

$$Access(H_0, Enf(pid_P, P)) \cap \mathcal{B} = \emptyset$$

(2) *For all user programs P_1 and P_2 in the subset J_{sub} with program ids pid_{P_1} and pid_{P_2} respectively*

$$Access_{l_g}(H_0, Enf(pid_{P_1}, P_1)) \cap Access_{l_g}(H_0, Enf(pid_{P_2}, P_2)) \subseteq \mathcal{P}_{nat} \cup \mathcal{P}_{noRen}$$

The proof of the above theorem is described in the Appendix A.

5 Case Study: FBJS

We studied the isolation mechanisms of FBJS and Yahoo! ADsafe because of their importance to hundreds of millions of Web users, and their relative simplicity. As reported in [17], we initially studied isolation based on filtering alone, and made suggestions for improvement in FBJS and ADsafe that have been adopted in both systems. However, the provably safe JavaScript subset based on filtering of [17] is far too restrictive to be used as a satisfactory replacement for FBJS.

In this paper, we therefore designed rewritings and wrapper functions to design a more expressive, provably safe subset of JavaScript. We believe that $J_{sub}(\mathcal{B})$ is now comparable to FBJS from the application developer viewpoint, has fewer semantic anomalies (as described below), and has the advantage of being provably safe.

Facebook. Facebook [7] is a well-known social networking Web site reporting 200 millions active users. Registered and authenticated users store private and public information on the Facebook website. Users can share information by sending messages, directly writing on a public portion of a user profile (called the wall), or interacting with Facebook applications. Facebook applications can be written by any user and can be deployed in various ways: as desktop applications, as external web pages displayed inside an `iframe` within a Facebook page, or as integrated components of a user profile.

Integrated Facebook applications are written in FBML [24], a variant of HTML designed to make it easy to write applications and also to restrict their possible behavior. A Facebook application is retrieved from the application publisher's server and embedded as a subtree of the Facebook page document. Since integrated Facebook applications are intended to interact with the rest of the user's profile, they are not isolated inside an `iframe`. As part of the Facebook isolation mechanism, the scripts used by applications must be written in a subset of JavaScript called FBJS [23] that restricts them from accessing arbitrary parts of the DOM tree of the larger Facebook page. The source application code is checked to make sure it contains valid FBJS, rewriting is applied to limit the application's behavior, and a specialized library is provided.

FBJS. While FBJS has the same syntax as JavaScript, a preprocessor consistently adds an application-specific prefix to all top-level identifiers in the code, isolating the effective namespace of an application from the namespace of other applicantions and of the rest of the Facebook page. For example, a statement document.domain may be rewritten to a12345_document.domain, where a12345_ is the application-specific prefix. This renaming will prevent application code from directly accessing most of the host and native JavaScript objects, such as the document object, Facebook provides libraries that are accessible within the application namespace. For example, the libraries include the object a12345_document, which mediates interaction between the application code and the true document object.

Additional steps are used to restrict the use of the this and o[e] in FBJS code. Occurrences of this are replaced with the expression $FBJS.ref(this), which calls the function $FBJS.ref to check what object this refers to when it is used. If this refers to window, then $FBJS.ref(this) returns null. FBJS rewrites o[e] to a12345_o[$FBJS.idx(e)], where $FBJS.idx enforces blacklisting on the string value of e.

Other, indirect ways that malicious content might reach the window object involve accessing certain standard or browser-specific predefined object properties such as __parent__ and constructor. Therefore, FBJS blacklists such properties and rewrites any explicit access to them in the code into an access to the useless property __unknown__. Finally, FBJS code runs in an environment where properties such as valueOf, which may access (indirectly) the window object, are redefined to something harmless.

Comparison. FBJS imposes essentially the same filtering restrictions as those we propose in Section 4, and the FBJS library appears to impose conditions similar to those we state in our wrapper conditions. However, there are some differences when it comes to renaming identifiers to place applications in separate namespaces and in the rewriting used to restrict this and e[e].

The renaming issue is that the FBJS implementation renames properties in the set $\mathcal{P}_{noRen}$ of properties we suggest should not be renamed. For example, toString() is rewritten to a12345_toString(), with an application-specific prefix. While toString() normally evaluates to *"[object Window]"*, the rewritten version throws a "reference error"

exception when evaluated. As noted in [17], FBJS does not correctly support renaming because it does not prevent explicit manipulation of the scope; the subset we propose here does not completely prevent access to scope objects either (for greater expressiveness), but has fewer pathological cases, because we avoid renaming $\mathcal{P}_{noRen}$ properties. A minor point is that we show that a safe subset can contain with, which FBJS prohibits, although our safe subset removes or restricts constructs that appear in many with use-cases.

To discuss more substantive issues, we consider $FBJS^{v}_{09}$, the version of FBJS deployed on Facebook at the time of our analysis, in March 2009. This version reflects repairs to the rewriting of this based on our earlier discovery of ways to redefine the run-time checking function [17]. The $FBJS^{v}_{09}$ $FBJS.ref function performs a check equivalent to NOGLOBAL, with some additional filtering to wrap DOM objects exposed to user code. Since $FBJS is effectively blacklisted in $FBJS^{v}_{09}$, we believe that ref prevents the this identifier from being evaluated to the window object; the check is semantically faithful to the requirements developed in Section 3.

On the other hand, the $FBJS^{v}_{09}$ $FBJS.idx function does not preserve the semantics of the property access, and as a result can be compromised in certain environments. More specifically, we report an attack we identified during the research reported here, a repair to prevent that attack, and a remaining problem. In the context of other filtering, $FBJS.idx is equivalent to

```
($=e2,($ instanceof Object||$blacklist[$])?"bad":$)
```

where $blacklist is the object {caller:true,$:true,$blacklist:true}. The main problem is that, in contrast to our definition of IDX, the expression $blacklist[$]?"bad":$ converts va to a string two times. This is a problem if evaluation has a side effect. For example, the object

```
{toString:function(){this.toString=function(){return "caller"}; return "good"}}
```

can fool FBJS by first returning a good property "good", and then returning the bad property "caller" on the second evaluation. To avoid this problem, $FBJS^{v}_{09}$ inserts the check $ instanceof Object that tries to detect if $ contains an object. In general, however, this check is not sound – according to the JavaScript semantics, any object with a null prototype (such as Object.prototype) escapes this check. Moreover, in Firefox, Internet Explorer and Opera the window object also escapes the check. In $FBJS^{v}_{09}$, Object.prototype and window are not accessible by user code, so cannot be used to implement this attack.

We found that the scope objects described in Section 3 have a null prototype in Safari, and therefore we were able to mount attacks on $FBJS.idx that effectively let user application code escape the Facebook sandbox. Shortly after we notified Facebook of this problem, the $FBJS.ref function was been modified to include a check of current browser, and if it is Safari an additional check that this is not bound to an object able to escape the instanceof check described above. This solution is not completely satisfactory, for two reasons. First, some browsers may have other host objects that have a null prototype, and that can be accesses without using this. Such objects could still be used to subvert $FBJS.idx, which has not been changed. Second, $FBJS.idx prevents objects from being uses as arguments of member expressions. This restriction is unnecessary for the safety of blacklisting, as shown by our proof for IDX.

6 Other Language-Based Approaches to Isolation

In this Section, we describe a few other approaches to JavaScript isolation which have not been subjected to rigorous semantic analysis, and could therefore benefit from the reasoning techniques presented in this paper. Due to space limitations, we do not discuss solutions based on idealized subsets of JavaScript with limited expressiveness, or that rely on browser modifications (for example [29]).

ADsafe. The Yahoo! ADsafe subset [5] is designed to allow advertising code to be placed directly on the host page, limiting interaction by a combination of static analysis and syntactic restrictions. The advertising code must satisfy very severe syntactical restrictions (including no this), and has access to an ADSAFE object, provided as a library, that mediates access to the DOM and other page services. Since we discovered that ADsafe was liable to prototype-poisoning attacks [15], the filtering process for ADsafe code has been complemented by a static analysis which gathers information about the objects that untrusted code may try to get access to. It is left to the page hosting the advertisement to make sure that those objects cannot be used to subvert the isolation mechanism. Our results show that some of the ADsafe restrictions are not strictly necessary, and the subset could be made more expressive.

BrowserShield. Browsershield is a system that rewrites web pages in order to enforce run-time monitoring of the embedded scripts. The systems takes an HTML page, adds a script tag to load a trusted library, rewrites embedded scripts so that they invoke a local rewriting function before being executed, and rewrites instructions to load remote scripts by making them load through a rewriting proxy. The run time monitoring is enforced by *policies* which are in effect functions that monitor the JavaScript execution. Common operations such as assignment suffer from a hundred-fold slowdown, and policies are arbitrary JavaScript functions for which there is no systematic way of guaranteeing correctness.

GateKeeper. Livshitz and Guarnieri [12] propose an approach to enforcing security and reliability policies in JavaScript based on static analysis based on two subsets. The first, $\mathrm{JS}_{\mathrm{Safe}}$, is obtained exclusively by filtering, and does not contain with, eval, e[e] or other dangerous constructs. The second subset, $\mathrm{JS}_{\mathrm{GK}}$ reinstates e[e] after wrapping it in a run-time monitor. The two subsets are amenable to static analysis by extracting Datalog facts and clauses that approximate the call-graph and points-to relation of JavaScript objects at run-time. The analysis necessarily loses precision in several points, and in particular when dealing with prototypes. Unfortunately, the implementation of GateKeeper is not available for inspection, and the sparse details on the definition of $\mathrm{JS}_{\mathrm{Safe}}$ and the run-time monitors in $\mathrm{JS}_{\mathrm{GK}}$ are not sufficient for a formal comparison with our results.

Caja. The Google Caja [4] project is a substantial effort to provide a safe JavaScript subset. Caja uses a compilation process that takes untrusted JavaScript and produces code in Cajita, a well behaved capability-based safe subset of JavaScript. Our goal is to isolate certain variables in the heap, whereas Caja enforces a finer grained security policy, which allows untrusted code from different principals to interact safely, by leveraging the capability-based paradigm. The Caja enforcement mechanisms also include filtering and rewriting, but the additional expressive power is gained at the price of complexity and efficiency. The reasoning techniques introduced in this paper

could be used to proof the correctness of such mechanisms, and possibly improve their implementations.

Lightweight Self-Protecting Javascript. Phung *et al.* [19] introduce a principled approach for enforcing safety properties on JavaScript native libraries. The enforcement mechanism involves wrapping each of the security critical native library methods and properties, before executing an untrusted script. Unfortunately, this approach is not sound for existing browsers. For example, by deleting certain properties of the global object, some native object are reinstated in the global environment, subverting the wrapping mechanism. Future versions of JavaScript may provide better support this implementation technique.

7 Conclusions

We systematically presented and analyzed a combination of isolation mechanisms for a subset of JavaScript that is comparable in expressiveness to Facebook FBJS [23]. Isolation from untrusted code in our subset of JavaScript is based on filtering out certain constructs (eval, Function, constructor), rewriting others (this, e1[e2]) to allow them to be used safely, and wrapping properties (*e.g.,* object and array prototype properties) of the execution environment to further limit the impact of untrusted code. Our analysis and security proofs build on a formal foundation for proving isolation properties of JavaScript programs [17], based on our operational semantics [15] of the full ECMA-262 Standard language (3rd Edition) [11]. While we focus on one particular combination of filters, rewriting functions, and wrappers, our methods are applicable to variants of the specific subset we present. For example, a DOM function such as createElement could be allowed, if suitable rewriting is used to insert checks on its string argument at run-time. In future work, we intend to examine Caja [4] and other systems, with the goal of providing provable security for practically useful language-based isolation mechanisms.

Acknowledgments. Mitchell and Taly acknowledge the support of the National Science Foundation. Maffeis is supported by EPSRC grant EP/E044956/1.

References

1. Irem Aktug, Mads Dam, and Dilian Gurov. Provably correct runtime monitoring. In *FM 2008*, volume 5014 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2008.
2. C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proc. of ECOOP'05*, page 429452, 2005.
3. A. Barth, C. Jackson, and J.C. Mitchell. Securing browser frame communication. In *17th USENIX Security Symposium*, 2008.
4. Google Caja Team. Google-Caja: A source-to-source translator for securing JavaScript-based web. http://code.google.com/p/google-caja/.
5. Douglas Crockford. ADsafe: Making JavaScript safe for advertising. http://www.adsafe.org/, 2008.
6. B. Eich. JavaScript at ten years. http://www.mozilla.org/js/language/ICFP-Keynote.ppt.
7. FaceBook. Web Site. http://www.facebook.com/.
8. D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 2006. http://proquest.safaribooksonline.com/0596101996.

9. P. Heidegger and P. Thiemann. Recency types for dynamically-typed, object-based languages. Foundations of Object-Oriented Languages (FOOL'09), 2009.
10. iGoogle. Web Site. http://www.google.com/ig.
11. ECMA International. ECMAScript language specification. stardard ECMA-262, 3rd Edition. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, 1999.
12. B. Livshits and S. Guarnieri. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. MSR-TR-2009-16, Feb. 2009.
13. S. Maffeis, J. Mitchell, and A. Taly. Complete ECMA 262-3 operational semantics. http://jssec.net/semantics/.
14. S. Maffeis, J.C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *Proc. of ESORICS'09*, LNCS.
15. S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS'08*, volume 5356 of *LNCS*, pages 307–325, 2008. See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.
16. S. Maffeis, J.C. Mitchell, and A. Taly. Run-time enforcement of untrusted javascript subsets. In *Web 2.0 Security & Privacy (W2SP)*, 2009.
17. S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.
18. OpenSocial. Web Site. http://www.opensocial.org/.
19. David Sands Phu H.Phung and Andrey Chudnov. Lightweight self protecting JavaScript. In *ASIACCS 2009*. ACM Press, 2009.
20. Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
21. C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of Dynamic HTML. *ACM Transactions on the Web*, 1(3), 2007.
22. A. Sabelfeld and Aslan Askarov. Tight enforcement of flexible information-release policies for dynamic languages. Second International Workshop on Proof-Carrying Code'08, 2008.
23. The FaceBook Team. FBJS. http://wiki.developers.facebook.com/index.php/FBJS.
24. The FaceBook Team. FBML. http://wiki.developers.facebook.com/index.php/FBML.
25. P. Thiemann. Towards a type system for analyzing javascript programs. In *Proc. of ESOP'05*, volume 3444 of *LNCS*, page 408422, 2005.
26. P. Thiemann. A type safe DOM API. In *Proc. of DBPL*, pages 169–183, 2005.
27. K. Vikram and M. Steiner. Mashup component isolation via server-side analysis and instrumentation. In *Web 2.0 Security & Privacy (W2SP)*, 2008.
28. YahooApp. Web Site. http://developer.yahoo.com/yap/.
29. D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proc. of POPL'07*, pages 237–249, 2007.

A Appendix: Formal Analysis

In this Section we prove the isolation theorem stated in Section 4. In order to prove this theorem, we make use of the operational semantics of JavaScript. The reader does not need to read this Appendix in order to understand the main body of the paper.

A.1 Preliminaries

We now extend some of the notation and properties of the operational semantics mentioned in Section 3.

As mentioned in Section 3, a *state* S is a triple (H, l, t). We use the notation $\mathcal{H}(S)$, $\mathcal{S}(S)$ and $\mathcal{T}(S)$ to denote each component of the state. We denote by H_0 the "empty" heap, that contains only the native objects, and no user code. We use l_g to denote the heap address of the global object #Global. If a heap, a scope pointer and a term are well-formed then the corresponding state is also well-formed (see the Appendix of [17] for a formal definition). In [15], we show that the evaluation of well-formed terms, if it terminates, yields either a value or an exception (for expressions), or a completion (for statements and programs). A state S is *initial* if it is well-formed, $\mathcal{H}(S) = H_0$, $\mathcal{S}(S) = l_g$ and $\mathcal{T}(S)$ is a user term. A *reduciton trace* τ is the (possibly infinite) maximal sequence of states $S_1, \ldots, S_n, \ldots$ such that $S_1 \rightarrow \ldots \rightarrow S_n \rightarrow \ldots$. Give a state S, we denote by $\tau(S)$ the (unique) trace originating from S and, if $\tau(S)$ is finite, we denote by $Final(S)$ the final state of $\tau(S)$. Given two states S_1 and S_2 such that $S_1 \rightarrow^* S_2$, we denote by $subTr(S_1, S_2)$ the trace of states from S_1 to S_2.

To ease our analysis, we add a separate sort mp to distinguish property names from strings and identifiers in the semantics. We make all the implicit conversions between these sorts explicit, by adding the identity functions Id2Prop: x → mp, Prop2Id: mp → x; Str2Prop: m → mp, Prop2Str: mp → m. The semantics already contained explicit conversion of strings to programs: ParseProg, ParseFunction, ParseParams. We also add context based reduction rules corresponding to each of the conversion functions, which apply to terms whose single step reduction involves an implicit conversion. In order to keep track of the names appearing in a state S, we define functions that collect respectively the identifiers and the property names of the term and the heap of S.

$$\mathcal{N}_I^T(S) = \{x | x \in \mathcal{T}(S)\} \quad \mathcal{N}_P^T(S) = \{mp \mid mp \in \mathcal{T}(S)\}$$
$$\mathcal{N}_I^H(S) = \{x \mid x \in P,\ P \in \mathcal{H}(S)\}$$
$$\mathcal{N}_P^H(S) = \{mp \mid \exists l : mp \in \mathcal{H}(S)(l)\}$$
$$\mathcal{N}_I(S) = \mathcal{N}_I^T(S) \cup \mathcal{N}_I^H(S) \quad \mathcal{N}_P(S) = \mathcal{N}_P^T(S) \cup \mathcal{N}_P^H(S)$$

Finally, we define the set of all the identifiers and property names appearing in a state S by $\mathcal{N}(S) = \mathcal{N}_I(S) \cup \text{Prop2Id}(\mathcal{N}_P(S))$. From these definitions, it follows that for any initial state S_0, $\mathcal{N}(S_0) = \mathcal{N}_I^T(S_0) \cup \mathcal{N}_P^H(S)$. $\mathcal{N}_P^H(S)$ is the set of property names present in the initial heap H_0. This is a fixed set, and will henceforth be denoted by $\mathcal{N}_P^0$.

We define *meta-call* a pair $(f, (args))$ where f is a semantic function or predicate appearing in the premise of a reduction rule, and $(args)$ is the list of its actual arguments as instantiated by a reduction step using that rule. For every state S, we denote by $\mathcal{C}_1(S)$ the set of the meta-calls triggered directly by a one step transition from state

S. Since each meta-call may in turn trigger other meta-calls during its evaluation, we denote by $\mathcal{C}(S)$ the set of all the meta-calls involved in a reduction step. We denote by $\mathcal{F}_H$ the set of functions that can read or write to the heap: $\mathcal{F}_H$ = {Dot(H, l, mp), Update(H, l,mp, v) }

For any state S, we define the set of all property names accessed during a single transition by $\mathcal{A}(S) \triangleq \{mp \mid \exists f \in \mathcal{F}_H \; \exists H, l : (f, (H, l, mp)) \in \mathcal{C}(S)\}$.

For any state S, we define the set of all property names of object at heap address l, accessed during a single transition by $\mathcal{A}_l(S) \triangleq \{mp \mid \exists f \in \{Dot, Update\} \; \exists H : (f, (H, l, mp)) \in \mathcal{C}(S)\}$.

In the case of a trace τ, $\mathcal{A}(\tau) \triangleq \bigcup_{S_i \in \tau} \mathcal{A}(S_i)$.

Definition 4. *(Access) Given a heap H and a program P, we define $Access(H, P)$ as $Access(H, P) = \mathcal{A}(\tau(H, l_g, P))$*

Definition 5. *(Access$_l$) Given a heap H, a program P and a heap address l, we define $Access(H, P)$ as $Access_l(H, P) = \mathcal{A}_l(\tau(H, l_g, P))$*

A.2 Proving the Isolation Theorem

In this section, we prove the isolation theorem which is restated below for convenience.

Restatement of Isolation theorem *Given a blacklist $\mathcal{B}$, such that $\mathcal{B} \cap \mathcal{P}_{nat} = \emptyset$, and the subset $J_{sub}(\mathcal{B})$, function Enf and the heap ${H_0}^{sub}$ as defined in Definitions 1, 2 and 3 respectively.*

(1) *For all user programs P in the subset $J_{sub}(\mathcal{B})$ with program ids pid_P*

$$Access(H_0, Enf(pid_P, P)) \cap \mathcal{B} = \emptyset$$

(2) *For all user programs P_1 and P_2 in the subset J_{sub} with program ids pid_{P_1} and pid_{P_2} respectively*

$$Access_{l_g}(H_0, Enf(pid_{P_1}, P_1)) \cap Access_{l_g}(H_0, Enf(pid_{P_2}, P_2)) \subseteq \mathcal{P}_{nat} \cup \mathcal{P}_{noRen}$$

We assume that the semantics of JavaScript is compliant with the ECMA-262 standard with the exception of the semantics of object literal. The ECMA-262 semantics for evaluation of object literals is to create a new empty object "as if by calling" new Object(). As suggested by various researchers, this is actually a bug in the semantics and the new object creation step should not execute new Object() explicitly. Instead a call to the "original" Object constructor should be *hard-wired* in the semantics. This is because in the case of an explicit call to new Object(), if the Object constructor is redefined or deleted by a malicious attacker then the whole object literal mechanism might fail. Thus in order to make the mechanism more robust, we consider the following modified semantic rule in our proofs:

$$\text{H,l,\{[(pn:e)}\tilde{}\text{]\}} \longrightarrow \text{H,l,@AddProps(new l\_Object()[,(pn:e)}\tilde{}\text{])} \; [\text{E}-\text{Obj}]$$

where l_{Object} is the heap address of the native Object constructor. Note that it is also possible to write safe rewriting function using the existing object literal rule by saving the original Object constructor in a $-variable, just like we save Function.prototype.call in

the expression IDX(e). We can then use this saved Object constructor to create new objects instead of using the object literal mechanism. We consider the modified semantic rule in our analysis mainly to make our presentation of the proof more clear.

In order to prove the theorem, we first define a state property $P^{pid}_{safe}(S)$ where S is any state obtained during corresponding the execution of a program with program id *pid*. This property states that the *single step reduction* of the state S does not involve accessing any property from the set $\mathcal{B}$ and all properties of the global object that are accessed during the reduction step are either in $\mathcal{P}_{noRen} \cup \mathcal{P}_{nat}$ or are prefixed with *pid*. A formal definition of this property is as follows.

Definition 6. *(P^{pid}_{safe}) Given a well-formed state S, $P^{pid}_{safe}(S)$ holds iff*

(1) $\mathcal{A}(S) \cap \mathcal{B} = \emptyset$
(2) $\forall mp \in \mathcal{A}_{l_g}(S) \setminus (\mathcal{P}_{nat} \cup \mathcal{P}_{noRen}) : isPrefix(pid, mp)$.

where for any strings m_1 and m_2, $isPrefix(m_1, m_2)$ is true iff m_1 is a prefix of m_2.

In order to prove the isolation theorem, it is sufficient to show that for each program P, the property $P^{pid_P}_{safe}(S)$ holds for all states S in $\tau(H_0, l_g, Enf(pid_P, P))$. We prove this by a defining a goodness property $Good^{pid}_{heap}(H)$ for a heap H and program id *pid* and goodness property $Good^{pid}_{term}(t)$ for a term t and program id *pid*. We combine the heap goodness and term goodness properties to define a property $Good^{pid}(S)$ for any wellformed state S and program id *pid*. We then show that the following hold:

(1) The initial heap ${H_0}^{sub}$ is good for any program id *pid* and $Good^{pid}_{term}(Enf(pid, P))$ is true for any user program P from the subset $J_{sub}(\mathcal{B})$ with program id *pid* (2) Given a program id *pid* and state S_1 such that $Good^{pid}(S_1)$ is true and S_1 is not a final state, S_1 can be reduced to (in possibly multiple steps) another state S_2 such that $Good^{pid}(S_2)$ is true and the safety property $P^{pid}_{safe}(S)$ holds for all states S between S_1 and S_2.

Before defining the goodness property, we state a few notations and definitions. Let $l_{Function}$, l_{String} denote the heap addresses of the constructors Function, String. Let l_{OP} and l_{AP} denote the heap address of the native object prototype and native array prototype. Let l_{eval}, l_{hOP}, l_{pIE} be the heap address of the methods eval, hasOwnProperty, propertyIsEnumerable. Let $l_{valueOf}$ and $l_{valueOfN}$ be the heap addresses of the original and wrapped valueOf method of Object.prototype. Similarly let l_{sort}, l_{sortN}; l_{concat}, $l_{concatN}$ and $l_{reverse}$, $l_{reverseN}$ be the heap addresses of the original and wrapped methods sort, concat and reverse of Array.prototype. Finally let l_{call} be the heap address of the call method of Function.prototype. We use the macro IDX1(e) to denote the expression

(l_g*$=e,{toString:function(){return ($=$String($),CHECK_$)}})

Term goodness $Good^{pid}_{term}$**.** : Given a program id *pid*, we say that $Good^{pid}_{term}(t)$ is true for a wellformed JS_{E2} user or internal term t iff it is contained in the subset $Terms^{pid}_{good}$ which is described using the grammar defined in Section A.3. The grammar is essentially a more restricted form of the JS_{E2} grammar. Certain production rules such as $e \rightarrow$ @cEval(P) are filtered and some other rules such as $e \rightarrow$ this and $e \rightarrow$ e[e] are rewritten to $e \rightarrow$ NOGLOBALTHIS and $e \rightarrow$ e[IDX(e)] respectively. For certain production rules in the grammar we also associate a predicate (also called *restriction*) on the *terminals* present in the derived sentence. These terminals include values, heap address,

property names and identifier names. A derived sentence is then *acceptable* only if it satisfies the corresponding predicate. For example the rule $e \rightarrow$ @Fun(l,e[,va~]) is associated with the predicate $l \notin \{l_{Function}, l_{eval}, l_{valueOf}, l_{sort}, l_{concat}, l_{reverse}\} \wedge va \notin \{l_{Function}, l_{eval}, l_{valueOf}, l_{sort}, l_{concat}, l_{reverse}, l_g\}$. So any derived term @Fun(l,e[,va~]) is acceptable only if l and va satisfy the predicate. In order to give the reader an intuitive understanding of the subset $Terms^{pid}_{good}$, we give the following informal definition for it. The subset $Terms^{pid}_{good}$ consists of all terms t such that

(1) Structure of t does not contain any property name or identifier from the set $\mathcal{B} \cup \{$eval, Function, constructor $\}$ except in contexts l_{hOP}.exe(l1,−), l_{pIE}(l1,−).
(2) All identifiers x appearing in an expression must have *pid* as a prefix.
(3) All sub-expressions of the form l_g∗mp must satisfy $mp \notin (\mathcal{P}_{noRen} \cup \mathcal{P}_{nat}) \Rightarrow mp \notin \mathcal{B} \wedge isPrefix(\$, mp)$.
(4) If the term contains a subexpression e1[e2] then e2 must be of one of the following:
   (a) IDX(e) for some expression e.
   (b) IDX1(e) for some expression e.
   (c) String m where m $\notin \mathcal{B} \cup$ {eval, Function, constructor }.
(5) If the term contain this then it must appear only inside subexpressions of the form NOGLOBALTHIS.
(6) Identifier $ only appears inside expressions of the form IDX(e) or IDX1(e) (for some expression e) or NOGLOBALTHIS.
(7) Structure of t does not contain any sub terms with contexts cEval() or FunParse().
(8) Structure of t does not contain any of the heap addresses $l_{Function}, l_{eval}, l_{valueOf}, l_{sort}, l_{concat}, l_{reverse}$.
(9) If heap address of the global object l_g is present in $\mathcal{T}(S)$ then it must appear inside one of the following contexts only: Function(fun([x~]){P},−); −.@Put(mp,va); l.@call(−,[va~]); l.@exe(−,[va~]); Fun(−,e[,va~]); @ExeFPA(l,−,va); @FunExe(−,P); @with(−ln1,ln2,s); −∗mp

where cEval(...) and FunParse(...) are *internal* continuation contexts which are entered during calls to the eval methods and Function constructor respectively.

Heap goodness ($Good^{pid}_{heap}$). : We now state a goodness property for wellformed heaps.

Definition 7. *Given a wellformed heap H and a program id pid, we say that $Good^{pid}_{heap}(H)$ is true iff:*

$$\begin{array}{l}
\forall l, p : H(l).p = l_{Function} \Rightarrow p = \textit{constructor} \\
\quad \vee \;\; p = \textit{Function} \\
\forall l, p : H(l).p = l_{eval} \Rightarrow p = \textit{eval} \\
\forall l, p : H(l).p = l_g \Rightarrow p = @scope \vee p = @this \vee p = @Fscope \\
\forall l, p : p \in H(l) \wedge isPrefix(\$, p) \Rightarrow l = l_g \\
\forall l, p : H(l).p = l_{valueOf} \Rightarrow l = l_g \wedge p = \$OPvalueOf \\
\forall l, p : H(l).p = l_{concat} \Rightarrow l = l_g \wedge p = \$APconcat \\
\forall l, p : H(l).p = l_{sort} \Rightarrow l = l_g \wedge p = \$APsort \\
\forall l, p : H(l).p = l_{reverse} \Rightarrow l = l_g \wedge p = \$APreverse \\
H(l_g).\$g = l_g \\
H(l_g).\$ = \text{""} \\
H(l_g).\$String = l_{String}
\end{array}$$

$H(l_{valueOf}).call = l_{call}$
$H(l_{valueOfN}).@body = function()\{var\ \$ = \$OPvalueOf.call(\ this,\ arguments);$
$return\ (\$==\$g?null:\$)\ \}$
$H(l_{valueOfN}).@Fscope = l_g$
$H(l_{sort}).call = l_{call}$
$H(l_{sortN}).@body = function(x)\{var\ \$ = \$APsort.call(\ this,\ arguments);$
$return\ (\$==\$g?null:\$)\ \}$
$H(l_{sortN}).@Fscope = l_g$
$H(l_{reverse}).call = l_{call}$
$H(l_{reverseN}).@body = function()\{var\ \$ = \$APreverse.call(\ this,\ arguments);$
$return\ (\$==\$g?null:\$)\ \}$
$H(l_{reverseN}).@Fscope = l_g$
$H(l_{concat}).call = l_{call}$
$H(l_{concatN}).@body = function(x\ )\{var\ \$ = \$APconcat.call(\ this,\ arguments);$
$return\ (\$==\$g?null:\$)\ \}$
$H(l_{concatN}).@Fscope = l_g$
$\forall l : \forall p \in H(l) : l \notin \{l_{valueOfN}, l_{reverseN}, l_{sortN}, l_{concatN}\} \Rightarrow \exists x, P :$
$(H(l).p = function(x)\{P\} \Rightarrow Good^{pid}_{term}(P) \wedge isPrefix(pid, x))$

Definition 8. *(State Goodness $Good^{pid}$) Given a program id pid and a wellformed state (H, l, t), we say that $Good^{pid}(H, l, t)$ is true iff $Good^{pid}_{heap}(H)$ and $Good^{pid}_{term}(t)$ are true.*

Main Results.

Lemma 1. *Given any program id pid, $Good^{pid}_{heap}({H_0}^{sub})$ is true and for any blacklist $\mathcal{B}$, for all programs in $P \in J_{sub}(\mathcal{B})$ with program id pid, $Good^{pid}_{term}(Enf(pid, P))$ is true.*

Proof. The heap ${H_0}^{sub}$ is defined as:

$${H_0}^{sub} = \mathcal{H}(Final(H_0, l_g, T_{idx}; T_{ng}; T_{valueOf}; T_{sort}; T_{concat}; T_{reverse}))$$

Since the reduction trace of the term $T_{idx}; T_{ng}; T_{valueOf}; T_{sort}; T_{concat}; T_{reverse}$ only consists of a few steps, it can be shown by performing symbolic execution and tracking all the heap updates, that the final heap obtained after executing the term
$T_{idx}; T_{ng}; T_{valueOf}; T_{sort}; T_{concat}; T_{reverse}$ is good.

We show that $\forall P \in J_{sub}(\mathcal{B}) : Good_{term} Enf(pid, P)$, by structural induction over the terms in $J_{sub}(\mathcal{B})$. For each term $Enf(pid, P)$, it can be shown that $Good^{pid}_{term}(Enf(pid, P))$ is true by showing that it is derivable from the grammar for $Terms^{pid}_{good}$ described in Section A.3.

Lemma 2. *Given a wellformed state S_1 and a program id pid such that $Good^{pid}(S_1)$ holds, either S_1 corresponds to a final state OR*

$$\exists S_2.S_1 \rightarrow^* S_2 \wedge Good^{pid}(S_2) \wedge \bigwedge_{S \in subTr(S_1, S_2)} P^{pid}_{safe}(S)$$

Proof. Given the program id pid, consider *any* good heap H and *any* wellformed scope address l. Proving the above lemma is equivalent to showing that:

$$\forall t \in Terms^{pid}_{good} : IsVal(t) \bigvee \exists S_2.((H,l,t)) \rightarrow^* S_2 \wedge Good^{pid}(S_2) \wedge \bigwedge_{S \in subTr(S_1,S_2)} P^{pid}_{safe}(S))$$

where $IsVal(t)$ is predicate which is true iff t denotes a final value.
We prove this by structural induction over the set of terms $Terms^{pid}_{good}$. In doing so we use the natural term structure obtained using grammar described in Subsection A.3.

Base Case. : The set of base case terms is

{ pv,l,r,co,w} ⊎
{ pv1 + pv2; x , l.@Call(vaw[,va~]) , l.@DefaultValue([String/Number]),
@PutValue(v,va) , @GetValue(r) , @ExeFPA(l,vaw,va) , l.@Put(l1,m,va)
PutLen(l,va), l[mp], ; , continue [x], break [x] } ⊎
{NOGLOBALTHIS}

(1) Each term in {pv,l,r,w, co} is a value from the set $Terms^{pid}_{good}$ and so the lemma holds trivially.

(2) Each term in { pv1 &+ pv2, x , l.@Call(vaw[,va~]) , l.@DefaultValue([String/Number]) , @PutValue(v,va) , @GetValue(r) , @ExeFPA(l,vaw,va) , l.@Put(l1,m,va) , l[mp], ;, break [x], continue [x], @PutValue(v,va);} except those in set T_{excpt} = {l_valueOfN.@Call(l[,va~]), l_sortN.@Call(l[,va~]), l_concatN.@Call(l[,va~]), l_reverseN.@Call(l[,va~])} has a transition axiom that applies to it. For each of these terms, we show that the state S_2 obtained by single step state transition of $S_1 = (H,l,t)$ has the property that $Good^{pid}(S_2)$ holds and also that $P^{pid}_{safe}(S_1)$ holds. We illustrate this on the term t = x as an example. The transition axiom that applies to x is [E−Ide−val] which is stated below:

$$\frac{\text{mp} = \text{convIdtoPname(x)} \quad \text{Scope(H,l,mp)=ln}}{\text{H,l,x} \longrightarrow \text{H,l,ln*mp}} \quad [\text{E−Ide−val}]$$

From the restriction on the term x we know that $g_3(x)$ and $isPrefix(pid,x)$. Therefore the corresponding property name mp also has the property $g_3(mp) \wedge isPrefix(pid,mp)$. It is easy to check that or all such mp, for every heap address ln, the reference ln*mp $\in Terms^{pid}_{good}$. Thus $Good^{pid}(H,l,ln*mp)$ holds. Since $\mathcal{A}(H,l,x) = \{@scope, @prototype\}$ $P^{pid}_{safe}(H,l,x)$ holds.

For each term in the set T_{excpt}, it follows by symbolic execution that $(H,l,t_I) \rightarrow^* H_2,l_2,l_3$ where l_3 is such that $l_3 \notin \{l_{Function}, l_{eval}, l_{valueOf}, l_{sort}, l_{concat}, l_{reverse}, l_g\}$. Also we show that $P^{pid}_{safe}(S)$ holds for all the intermediate states.

(3) For the term NOGLOBALTHIS, it follows by symbolic execution with respect to a good heap H that $(H,l,\text{NOGLOBALTHIS}) \rightarrow^* (H_2,l_2,l_3)$ where $l_3 \notin \{l_{Function}, l_{eval}, l_{valueOf}, l_{sort}, l_{concat}, l_{reverse}, l_g\}$. Also we show that $P^{pid}_{safe}(S)$ holds for all the intermediate states S.

Inductive Case. : It is possible to split the set of inductive cases as $T^{indB} \uplus T^{indC}$ where T^{indB} denotes the set of terms to which a transition axiom applies and T^{indC} is the set of terms which can be expressed as $C_1(t)$ where C_1 is a valid evaluation context

from the operational semantics [15] and t is a well-formed JS_{E2} user or internal term. After analyzing all the inductive cases, we obtain the following definitions for the sets T^{indB} and T^{indC}.

$$T^{indB} = \left\{ \begin{array}{l} \text{"\{"[(pn:e)~]"\}"} \text{ ;do s while "("e")"; while "("e")"s ;} \\ \text{@Typeof(e); @"<"S(b,e,e); @"<"N(b,e,e)} \\ \text{@L(b,va1,va2,e) ; var [(x["="e])~]; function x "("[x~]")"\{"[P] "\}"} \\ \text{"\{"s*"\}"; l.@Construct([e,][va~]); @ConstructCall(l,e)} \\ \text{for "("e in e")"s; e.x; for "("var x["="e] in e")"s; "["[(e|,)~] "]"} \end{array} \right\}$$

$$T^{indC} = \left\{ \begin{array}{l} \text{@TS(e); @TN(e); @TP(e); @GV(e) ; @AddProps(e,[,(pn:e)~]); @Fun(l,e[,va~]);} \\ \text{@FunExe(l,P); l.@Construct([e,][va~]); @ConstructCall(l,e);} \\ \text{"("e")"; e"["IDX(e)"]"; e"["IDX1(e)"]"; new e["("[e~]")"];} \\ \text{e"("[e~]")"; e PO ; UN e; e BIN e; "("e"?"e":"e")"; @VarList([e,][x[=e]~]);} \\ \text{"("e","e")"; e; if "("e")"s [else s]; return [e]; @PO(v,e,n); l.@Exe(l1[,e~]);} \\ \text{with "("e")"s; id:s; throw e; try "\{"s*"\}"[catch "("x")""\{"s1*"\}"];} \\ \text{[finally "\{"s2*"\}"]; @while(e,s,ls,vae,e,s); @Block(co[,s+]);} \\ \text{@with (l,ln1,ln2,s); @eforin(e,ls,s,vae,l,e,m); @pforin(e,ls,s,vae,l,m);} \\ \text{@sforin(e,ls,s,vae,l,s,m); ls>s; fd [P]; s[P]; @ArrayLiteral(e,n,([e~]));} \\ \text{@GetDefault(l,m,e) ; l.@Construct([e,][va~]); @ConstructCall(l,e)} \end{array} \right\}$$

Note that the terms in the set T^{indB} and T^{indC} are *good* only under the restriction mentioned in the grammar defined in Subsection A.3. Henceforth we will assume that all terms in T^{indB} and T^{indC} satisfy the restriction predicates.

(1) T^{indB} : A transition axiom applies to each term $t_I \in T^{indB}$. For each term, we show that the state S_2 obtained by single step state transition of $S_1 = (H, l, t_I)$ has the property that $Good^{pid}(S_2)$ holds and also that $P^{pid}_{safe}(S_1)$ holds. In showing this property we make use of the fact that $t_I \in Terms^{pid}_{good}$ and $Good^{pid}_{heap}(H)$ hold. We illustrate this argument on the term $t_I =$ "{"[(pn:e)~]"}" as an example. The transition axiom that applies to "{"[(pn:e)~]"}" is [E−Obj] which is stated below:

$$\text{H,l,\{[(pn:e)~]\}} \longrightarrow \text{H,l,@AddProps(new l\_Object() [,(pn:e)~])}$$

From the grammar, it is easy to see that @AddProps(e,[,(pn:e)~])} is derivable and hence
AddProps(new l_Object()[,(pn:e)~]) $\in Terms^{pid}_{good}$. Therefore,
$Good^{pid}(H, l,$ AddProps(new l_Object()[,(pn:e)~])) is true.
Since $\mathcal{A}(H, l,$ AddProps(new l_Object()[,(pn:e)~])) $= \emptyset$, we conclude that
$P^{pid}_{safe}(H, l,$ AddProps(new l_Object()[,(pn:e)~])) is trivially true. Thus the lemma holds for $t_I =$ "{"[(pn:e)~]"}".

(2) T^{indC}: The term t_I can be expressed as $t_I = C_1(t)$ where C_1 is some evaluation context defined in our operational semantics and t is a wellformed JS_{E2} user or internal term. It turns that all terms in T^{indC} can be expressed as $C_1(t)$ with the additional property that $t \in Terms^{pid}_{good}$. Since t is a sub-term of t_I and $t \in Terms^{pid}_{good}$, we can apply the induction hypothesis on t. This gives us the following two cases:

(a) t **does *not* correspond to a final value:** In this case by the induction hypothesis we have for state $S_1 = (H, l, t)$,
$\exists S_2 : S_1 \rightarrow^* S_2 \wedge Good^{pid}(S_2) \wedge \bigwedge_{S \in subTr(S_1,S_2)} P^{pid}_{safe}(S)$
Given a state $S = (H, l, t)$, we define $C(S) = (H, l, C(t))$. The general form of a context rule is therefore $\frac{S \rightarrow S'}{C(S) \rightarrow C(S')}$ where C is an evaluation context defined in the semantics.
Applying the context rule multiple times to state $C_1(S_1) = (H, l, C_1(t_1))$, we get $C_1(S_1) \rightarrow^* C_1(S_2)$. Let $S_2 = (H_2, l_2, t_2)$. Since $Good^{pid}(H_2, l_2, t_2)$ holds, $Good^{pid}_{heap}(H_2)$ holds, l_2 is a well-formed scope address and $t_2 \in Terms^{pid}_{good}$. Therefore $C(t_2) \in Terms^{pid}_{good}$ and as a result $Good^{pid}(H_2, l_2, C(t_2))$ holds. Using the definition of P_{safe}, it is easy to show that $\bigwedge_{S \in subTr(S_1,S_2)} P^{pid}_{safe}(S) \Rightarrow \bigwedge_{S \in subTr(C(S_1),C(S_2))} P^{pid}_{safe}(S)$. This is essentially because for any state $C(S)$ where S is not a final state and C is an evaluation context, $\mathcal{A}(C(S)) = \mathcal{A}(S)$ and $\forall l : \mathcal{A}_l(C(S)) = \mathcal{A}_l(S)$. Thus we have
$C(S_1) \rightarrow^* C(S_2) \wedge Good^{pid}(C(S_2)) \wedge \bigwedge_{S \in subTr(C(S_1),C(S_2))} P^{pid}_{safe}(S)$
Thus the lemma is true for this case.

(b) t **corresponds to a *good* value** v**:** In this case we have $t_I = C_1(v)$. For all such terms either a transition axiom or a context rule applies to it. After analyzing all terms in T^{indC}, we find that for all terms other than $\{$va1[IDX(e2)], va1[IDX1(va2)]$\}$, the same argument as mentioned in cases 1 and 2a applies, that is, we can either
express the term $C_1(v)$ as $C_1(v) = C_2(t_2)$ where t_2 is not a final value OR
show that a transition axiom applies to $C_1(v)$ and a single step reduction under the axiom is safe and leads to a good term. In the remaining part we explain the analysis for the terms $\{$va1[IDX(e2)], va1[IDX1(va2)]$\}$:

- va1[IDX(e2)]: Although va1[IDX(e2)] can be expressed as C[IDX(e2)] for the evaluation context $C =$ va1[$-$], we cannot use the argument from case 2a because the term IDX(e2) is not in $Terms^{pid}_{good}$. However from the operational semantics, we know that a single step transition of the state $(H, l, va1[IDX(e2)])$, where H is a good heap, moves it to the state $(H, l, va1[IDX1(e2)])$.

$$(H, l, va1[IDX(e2)]) \rightarrow (H, l, va1[IDX1(e2)]) \text{Rule [E−Ide−val]}$$

  Now from the grammar we know that va1[IDX1(e2)] $\in Terms^{pid}_{good}$. Since $\mathcal{A}(H, l, va1[IDX(e2)]) = \{@scope, @prototype\}$ $P^{pid}_{safe}(H, l, va1[IDX(e2)])$ holds. Thus the lemma is true for this case.
- va1[IDX1(va2)]$\}$: For this case, it follows by symbolic execution with respect to a good heap H that $(H, l,$ va1[IDX1(va2)]$) \rightarrow^* H_2, l_2,$ l_3[mp]) where mp is such that
  $g_3(mp) \wedge \neg(isPrefix(\$, x))$ and
  $l_3 \notin \{l_{Function}, l_{eval}, l_{valueOf}, l_{sort}, l_{concat}, l_{reverse}, l_g\}$.
  Also we show that $P^{pid}_{safe}(S)$ holds for all the intermediate states.

□

Proof Sketch of Isolation theorem:. Follows from Lemma 1 and Lemma 2 □

A.3 Grammar for $Terms^{pid}_{good}$

Given a program id pid, we define a restricted context free grammar for the set of terms $Terms^{pid}_{good}$. The table below gives the set of production rules for the nonterminals e, s and P corresponding to expressions, statements and programs. The third column of the table specifies restrictions on values, property names, identifier names and heap addresses appearing in the derived term. To shorten the description, we use the following macros.

$g_1(l) = l \notin \{l_{Function}, l_{eval}, l_{valueOf}, l_{sort}, l_{concat}, l_{reverse}, l_g\}$

$g_2(l) = l \notin \{l_{Function}, l_{eval}, l_{valueOf}, l_{sort}, l_{concat}, l_{reverse}\}.$

$g_3(x) = x \notin \mathcal{B} \cup \{constructor, eval, function\}$

where g_3 is also defined for property names mp and strings m.

Good Expressions.

e ::=	l	$g_1(l)$
	r	$r = l * mp \Rightarrow$ $((g_3(mp) \wedge \neg(isPrefix(\$, mp)))\wedge$ $((l = l_g \wedge mp \notin (\mathcal{P}_{noRen} \cup \mathcal{P}_{nat})) \Rightarrow$ $isPrefix(pid, mp)))$
	w	$w = \lvert l \rvert \Rightarrow g_1(l)$
	pv1 &+ pv2	
	@AddProps(e,[,(pn:e)~])	$pn \notin \mathcal{B} \cup \{constructor, eval, function\}$ $\wedge\neg(isPrefix(\$, pn))$
	@TS(e)	
	@TN(e)	
	@TP(e)	
	@GV(e)	
	@Fun(l,e[,va~])	$g_1(va) \wedge g_2(l)$
	@PO(v,e,n)	$g_1(v) \bigwedge v = l * mp \Rightarrow$ $((g_3(mp) \wedge \neg(isPrefix(\$, mp)))\wedge$ $((l = l_g \wedge mp \notin (\mathcal{P}_{noRen} \cup \mathcal{P}_{nat})) \Rightarrow$ $isPrefix(pid, mp)))$
	@Typeof(e)	
	@"<"S(b,e,e)	
	@"<"N(b,e,e)	
	@L(b,va1,va2,e)	$g_1(va1) \wedge g_1(va2)$
	@ArrayLiteral(e,n,([e~]))	
	l.@Call(vaw[,va~])	$g_1(l) \wedge g_2(vaw) \wedge g_1(va)$
	l.@Exe(l1[,e~])	$g_1(l) \wedge g_2(l1)$
	@FunExe(l,P)	
	l.@Construct([e,][va~])	$g_1(va) \wedge g_1(l)$
	@ConstructCall(l,e)	$g_1(l)$
	l.@DefaultValue([String])	$g_1(l)$
	l.@DefaultValue(Number)	$g_1(l)$
	@GetDefault(l,m,e)	$g_1(l) \wedge (g_3(m) \wedge \neg(isPrefix(\$, x)))$

e ::=	@PutValue(v,va)	$g_1(va) \wedge g_1(v) \wedge v = l * mp \Rightarrow$ $((g_3(mp) \wedge \neg(isPrefix(\$, mp))) \wedge$ $((l = l_g \wedge mp \notin (\mathcal{P}_{noRen} \cup \mathcal{P}_{nat})) \Rightarrow$ $isPrefix(pid, mp))) \bigwedge$
	@GetValue(v)	$g_1(v) \bigwedge v = l * mp \Rightarrow$ $((g_3(mp) \wedge \neg(isPrefix(\$, mp))) \wedge$ $((l = l_g \wedge mp \notin (\mathcal{P}_{noRen} \cup \mathcal{P}_{nat})) \Rightarrow$ $isPrefix(pid, mp)))$
	@ExeFPA(l,vaw,va)	$g_1(l) \wedge g_2(vaw) \wedge g_1(va)$
	l.@Put(m,va)	$g_2(l) \wedge g_1(va) \wedge (g_3(m) \wedge \neg(isPrefix(\$, x)))$
	@PutLen(l,va)	$g_2(l) \wedge g_1(va)$
	NOGLOBALTHIS	
	x	$g_3(x) \wedge isPrefix(pid, x)$
	pv	
	"[" [(e|,)~] "]"	
	"{" [(pn:e)~] "}"	
	"(" e ")"	
	e.x	$g_3(x) \wedge \neg(isPrefix(\$, x))$
	e "[" IDX(e) "]"	
	e "[" IDX1(e) "]"	
	e "[" mp "]"	$g_3(mp) \wedge \neg(isPrefix(\$, x))$
	new e["(" [e~] ")"]	
	e "(" [e~] ")"	
	function [x] "(" [x~] "){" [P] "}"	$g_3(x) \wedge isPrefix(pid, x)$
	e &PO	
	&UN e	
	e &BIN e	
	"(" e "?" e ":" e ")"	
	"(" e "," e ")"	

Good Statements.

s ::=		
	"{" s* "}"	
	var [(x["="e])~]	$g_3(x) \wedge isPrefix(pid, x)$
	;	
	e	
	if "(" e ")" s [else s]	
	while "(" e ")" s	
	do s while "(" e ")"	
	for "(" e in e ")" s	
	for "(" var x["="e] in e ")" s	$g_3(x) \wedge isPrefix(pid, x)$
	continue [x]	
	break [x]	
	return [e]	
	with "(" e ")" s	
	id:s	
	throw e;	

s ::=		
	try "{"s∗"}"[catch "("x"){"s1∗"}"] [finally "{"s2∗"}"]	$g_3(x) \wedge isPrefix(pid, x)$
	co	$g_1(val(co))$
	@while(e,s,ls,vae,e,s)	$g_1(vae)$
	@Block(co[,s+])	$g_1(val(co))$
	@VarList([e,][x[=e]˜])	$g_3(x) \wedge isPrefix(pid, x)$
	@with (l,ln1,ln2,s)	$g_2(l) \wedge g_2(ln1) \wedge g_2(ln2)$
	@eforin(e,ls,s,vae,l,e,m)	$g_1(l) \wedge g_1(vae)$
	@pforin(e,ls,s,vae,l,m)	$g_1(l) \wedge g_1(vae)$
	@sforin(e,ls,s,vae,l,s,m)	$g_1(l) \wedge g_1(vae)$
	ls>s	

Good Programs.

P ::=		
	fd [P]	
	s[P]	

fd ::=		
	function x "("[x˜]"){"[P]"}"	$g_3(x) \wedge isPrefix(pid, x)$