Transaction-based Sandboxing for JavaScript
Technical Report

Matthias Keil and Peter Thiemann

University of Freiburg
Freiburg, Germany
`{keilr,thiemann}@informatik.uni-freiburg.de`

Abstract

Today's JavaScript applications are composed of scripts from different origins that are loaded at run time. As not all of these origins are equally trusted, the execution of these scripts should be isolated from one another. However, some scripts must access the application state and some may be allowed to change it, while preserving the confidentiality and integrity constraints of the application.

This paper presents design and implementation of DecentJS, a language-embedded sandbox for full JavaScript. It enables scripts to run in a configurable degree of isolation with fine-grained access control. It provides a transactional scope in which effects are logged for review by the access control policy. After inspection of the log, effects can be committed to the application state or rolled back.

The implementation relies on JavaScript proxies to guarantee full interposition for the full language and for all code, including dynamically loaded scripts and code injected via ***eval***. Its only restriction is that scripts must be compliant with JavaScript's strict mode.

1998 ACM Subject Classification D.4.6 Security and Protection

Keywords and phrases JavaScript, Sandbox, Proxy

1 Introduction

JavaScript is used by 93.1%[1] of all the websites. Most of them rely on third-party libraries for connecting to social networks, feature extensions, or advertisement. Some of these libraries are packaged with the application, but others are loaded at run time from origins of different trustworthiness, sometimes depending on user input. To compensate for different levels of trust, the execution of dynamically loaded code should be isolated from the application state.

Today's state of the art in securing JavaScript applications that include code from different origins is an all-or-nothing choice. Browsers apply protection mechanisms, such as the same-origin policy [33] or the signed script policy [36], so that scripts either run in isolation or gain full access.

While script isolation guarantees noninterference with the working of the application as well as preservation of data integrity and confidentiality, there are scripts that must have access to part of the application state to function meaningfully. As all included scripts run with the same authority, the application script cannot exert fine-grained control over the use of data by an included script.

Thus, managing untrusted JavaScript code has become one of the key challenges of present research on JavaScript [2, 15, 5, 6, 32, 25, 30, 24, 23, 12]. Existing approaches are either based on restricting JavaScript code to a statically verifiable language subset (e.g.,

[1] according to http://w3techs.com/, status as of March 2016

Facebook's FBJS [8] or Yahoo's ADsafe [1]), on enforcing an execution model that only forwards selected resources into an otherwise isolated compartment by filtering and rewriting like Google's Caja project [11], or on implementing monitoring facilities inside the JavaScript engine [32].

However, these approaches have known deficiencies: the first two need to restrict usage of JavaScript's dynamic features, they do not apply to code generated at run time, and they require extra maintenance efforts because their analysis and transformation needs to be kept in sync with the evolution of the language. Implementing monitoring in the JavaScript engine is fragile and incomplete: while efficient, such a solution only works for one engine and it is hard to maintain due to the high activity in engine development and optimization.

Contributions

We present the design and implementation of DecentJS, a sandbox for JavaScript which enforces noninterference (integrity and confidentiality) by run-time monitoring. Its design is inspired by revocable references [39, 26] and SpiderMonkey's compartment concept [41].

Compartments create a separate memory heap for each website, a technique initially introduced to optimize garbage collection. All objects created by a website are only allowed to touch objects in the same compartment. Proxies are the only objects that can cross the compartment boundaries. They are used as cross compartment wrappers to make objects accessible in other compartments.

DecentJS adapts SpiderMonkey's compartment concept. Each sandbox implements a fresh scope to run code in isolation to the application state. Proxies implement a membrane [39, 26] to guarantee full interposition and to make objects accessible inside of a sandbox.

Outline of this Paper

The paper is organized as follows: Section 2 introduces DecentJS's facilities from a programmer's point of view. Section 3 recalls proxies and membranes from related work and explains the principles underlying the implementation. Section 4 discusses DecentJS's limitations and Section 5 reports our experiences from applying sandboxing to a set of benchmark programs. Finally, Section 6 concludes.

Appendix A presents an example demonstrating the sandbox hosting a third-party library. Appendix B shows some example scenarios that already use the implemented system. Appendix C shows the operational semantics of a core calculus with sandboxing. Appendix D states some technical results. Appendix E discusses related work and Appendix **??** reports our experiences from applying sandboxing to a set of benchmark programs.

2 Transaction-based Sandboxing: A Primer

This section introduces transaction-based sandboxing and shows a series of examples that explains how sandboxing works.

Transactional sandboxing is inspired by the idea of transaction processing in database systems [43] and transactional memory [35]. Each sandbox implements a transactional scope the content of which can be examined, committed, or rolled back.

Central to our sandbox is the implementation of a membrane on values that cross the sandbox boundary. The membrane supplies effect monitoring and guarantees noninterference. Moreover, it features identity preservation and handles shadow objects. *Shadow objects* allow sandbox-internal modifications of objects without effecting there origins. The modified version

```
function Node (value, left, right) {
  this.value = value;
  this.left = left;
  this.right = right;
}
Node.prototype.toString = function () {
  return (this.left?this.left + ", ":"") + this.value +(this.right?", "+this.right:"");
}
function heightOf (node) {
  return Math.max(((node.left)?heightOf(node.left)+1:0), ((node.right)?heightOf(node.
      right)+1:0));
}
function setValue (node) {
  if (node) {
    node.value=heightOf(node);
    setValue(node.left);
    setValue(node.right);
  }
}
```

Figure 1 Implementation of *Node*. Each node object consists of a value field, a left node, and a right node. Its prototype provides a *toString* method that returns a string representation. Function *heightOf* computes the height of a node and function *setValue* replaces the value field of a node by its height, recursively.

is only visible inside of the sandbox and different sandbox environments may manipulate the same object.

Sandboxing provides transactions, a unit of effects that represent the set of modifications (write effects) on its membrane. Effects enable to check for conflicts and differences, to rollback particular modifications, or to commit a modification to its origin.

The implementation of the system is available on the web[2].

2.1 Cross-Sandbox Access

We consider operations on binary trees as defined by *Node* in Figure 1 along with some auxiliary functions. As an example, we perform operations on a tree consisting of one node and two leaves. All value fields are initially *0*.

```
var root = new Node(0, new Node(0), new Node(0));
```

Next, we create a new empty sandbox by calling the constructor *Sandbox*. Its first parameter acts as the global object of the sandbox environment. It is wrapped in a proxy to mediate all accesses and it is placed on top of the scope chain for code executing inside the sandbox. The second parameter is a configuration object. A sandbox is a first class value that can be used for several executions.

```
var sbx = new Sandbox(this, {/* some parameters */});
```

[2] https://github.com/keil/DecentJS

One use of a sandbox is to wrap invocations of function objects. To this end, the sandbox API provides methods *call*, *apply*, and *bind* analogous to methods from *Function.prototype*. For example, we may call *setValue* on *root* inside of *sbx*.

```
21 sbx.call(setValue, this, root);
```

The first argument of *call* is a function object that is decompiled and redefined inside the sandbox. This step erases the function's free variable bindings and builds a new closure relative to the sandbox's global object. The second argument, the receiver object of the call, as well as the actual arguments of the call are wrapped in proxies to make these objects accessible inside of the sandbox.

The wrapper proxies mediate access to their target objects outside the sandbox. Reads are forwarded to the target unless there are local modifications. The return values are wrapped in proxies, again. Writes produce a *shadow value* (cf. Section 3.2) that represents the sandbox-internal modification of an object. Initially, this modification is only visible to reads inside the sandbox.

Native objects, like the *Math* object in line 10, are also wrapped in a proxy, but their methods cannot be decompiled because there exists no string representation. Thus, native methods must either be trusted or forbidden. Fortunately, most native methods do not have side effects, so we chose to trust them.

Given all the wrapping and sandboxing, the call in line 21 did not modify the *root* object:

```
22 root.toString(); // returns 0, 0, 0
```

But calling *toString* inside the sandbox shows the effect.

```
23 sbx.call(root.toString, root); // return 0, 1, 0
```

2.2 Effect Monitoring

During execution, each sandbox records the effects on objects that cross the sandbox membrane. The resulting lists of *effect objects* are accessible through *sbx.effects*, *sbx.readeffects*, and *sbx.writeeffects* which contain all effects, read effects, and write effects, respectively. All three lists offer query methods to select the effects of a particular object.

```
24 sbx.call(heightOf, this, root);
25 var rects = sbx.effectsOf(this);
26 print(";;; Effects of this");
27 rects.foreach(function(i, e) {print(e)});
```

The code snippet above prints a list of all effects performed on *this*, the global object, by executing the *heightOf* function on *root*. The output shows the resulting accesses to *heightOf* and *Math*.

```
28 ;;; Effects of this
29 (1425301383541) has [name=heightOf]
30 (1425301383541) get [name=heightOf]
31 (1425301383543) has [name=Math]
32 (1425301383543) get [name=Math]
33 ...
```

The first column shows a timestamp, the second shows the name of the effect, and the last column shows the name of the requested parameter. The list does not contain write accesses to *this*. But there are write effects to *value* from the previous invocation of *setValue*.

```
var wectso = sbx.writeeffectsOf(root);
print(";;; Write Effects of root");
wectso.foreach(function(i, e) {print(e)});
```

```
;;; Write Effects of root
(1425301634992) set [name=value]
```

2.3 Inspecting a Sandbox

The state inside and outside of a sandbox may diverge for different reasons. We distinguish changes, differences, and conflicts.

A *change* indicates if the sandbox-internal value has been changed with respect to the outside value. A *difference* indicates if the outside value has been modified after the sandbox has concluded. For example, a difference to the previous execution of *setValue* arises if we replace the left leaf element by a new subtree of height 1 outside of the sandbox.

```
root.left = new Node(new Node(0), new Node(0));
```

Changes and differences can be examined using an API that is very similar to the effect API. There are flags to check whether a sandbox has changes or differences as well as iterators over them.

A *conflict* arises in the comparison between different sandboxes. Two sandbox environments are in conflict if at least one sandbox modifies a value that is accessed by the other sandbox later on. We consider only Read-After-Write and Write-After-Write conflicts. To demonstrate conflicts, we define a function *appendRight*, which adds a new subtree on the right.

```
function appendRight (node) {
  node.right = Node('a', Node('b'), Node('c'));
}
```

To recapitulate, the global *root* is still unmodified and prints *0,0,0,0,0*, whereas the *root* in *sbx* prints *0,0,0,1,0*. Now, let's execute *appendRight* in a new sandbox *sbx2*.

```
var sbx2 = new Sandbox(this, {/* some parameters */});
sbx2.call(appendRight, this, root);
```

Calling *toString* in *sbx2* prints *0,0,0,0,b,a,c*. However, the sandboxes are *not* in conflict, as the following command show.

```
sbx.inConflictWith(sbx2); // returns false
```

While both sandboxes manipulate *root*, they manipulate different fields. *sbx* recalculates the field *value*, whereas *sbx2* replaces the field *right*. Neither reads data that has previously been written by the other sandbox. However, this situation changes if we call *setValue* again, which also modifies *right*.

```
sbx.call(setValue, this, root);
var cofts = sbx.conflictsWith(sbx2); // returns a list of conflicts
cofts.foreach(function(i, e) {print(e)});
```

It documents a read-after-write conflict:

```
Confict: (1425303937853) get [name=right]@SBX001 – (1425303937855) set [name=
    right]@SBX002
```

2.4 Transaction Processing

The *commit* operation applies select effects from a sandbox to its target. Effects may be committed one at a time by calling *commit* on each effect object or all at once by calling *commit* on the sandbox object.

```
sbx.commit();
root.toString(); // returns 0, 1, 0, 2, 0
```

The *rollback* operation undoes an existing manipulation and returns to its previous configuration before the effect. Again, rollbacks can be done on a per-effect basis or for the sandbox as a whole. However, a rollback did not remove the shadow object. Thus, after rolling back, the values are still shadow values in *sbx*.

```
sbx.rollback();
root.toString(); // returns 0, 1, 0, 2, 0
sbx.call(toString, this, root); // returns 0, 0, 0, 0, 0
```

The *revert* operation resets the shadow object of a wrapped value. The following code snippet reverts the *root* object in *sbx*.

```
sbx.revertOf(root);
```

Now, *root*'s shadow object is removed and the origin is visible again in the sandbox. Calling *toString* inside of *sbx* returns *0,1,0,2,0*.

3 Sandbox encapsulation

The implementation of DecentJS builds on two foundations: *memory safety* and *reachability*. In a memory safe programming language, a program cannot access uninitialized memory or memory outside the range allocated to a datastructure. An object reference serves as the right to access the resources managed by the object along with the memory allocated to it. In JavaScript, all resources are accessible via property read and write operations on objects. Thus, controlling reads and writes is sufficient to control the resources.

DecentJS ensures isolation of the actual program code by intercepting each operation that attempts to modify data visible outside the sandbox. To achieve this behavior, all functions and objects crossing the sandbox boundary are wrapped in a *membrane* to ensure that the sandboxed code cannot modify them in any way. This membrane is implemented using JavaScript proxies [39].

More precisely, our implementation of sandboxing is inspired by *Revocable Membranes* [39, 37] and access control based on object capabilities [28]. Identity preserving membranes keep the sandbox apart from the normal program execution: We encapsulate objects passed through the membrane and redirect write operations to shadow objects (Section 3.2), we encapsulate code (Section 3.3), and we withhold external bindings from a function (Section 3.4). No unprotected value is passed inside the sandbox.

3.1 Proxies and membranes

A *proxy* is an object intended to be used in place of a *target object*. The proxy's behavior is controlled by a *handler object* that typically mediates access to the target object. Both, target and handler, may be proxy objects themselves.

The handler object contains trap functions that are called when a trapped operation is performed on the proxy. Operations like property read, property write, and function

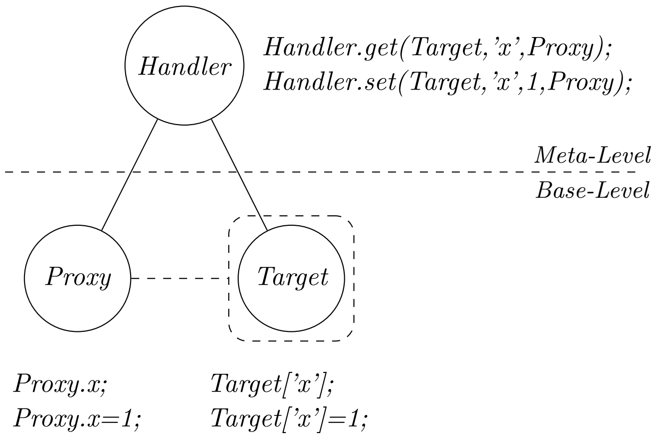


Figure 2 Proxy operations. The operation *Proxy.x* invokes the trap *Handler.get(Target,'x',Proxy)* (property get) and the property set operation *Proxy.x=1* invokes *Handler.set(Target,'x',1,Proxy)*.

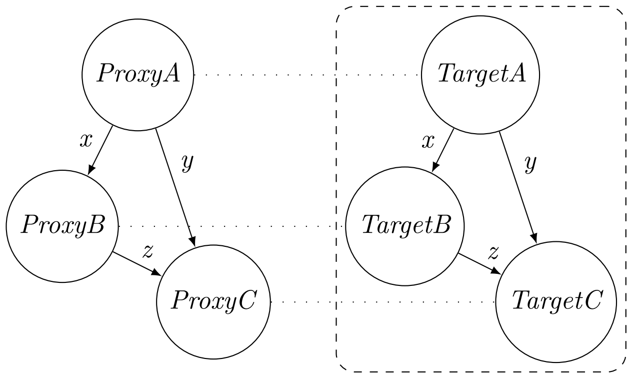


Figure 3 Property access through an identity preserving membrane (dashed line around target objects). The property access through the wrapper *ProxyA.x* returns a wrapper for *TargetA.x*. The property access *ProxyA.y* returns the same wrapper as *ProxyB.z*.

application are forwarded to their corresponding trap. The trap function may implement the operation arbitrarily, for example, by forwarding the operation to the target object. The latter is the default behavior if the trap is not specified.

Figure 2 illustrates this situation with a handler that forwards all operations to the target.

A *membrane* is a regulated communication channel between an object and the rest of the program. A membrane is implemented by a proxy that guards all operations on its target. If the result of an operation is another object, then it is recursively wrapped in a membrane before it is returned. This way, all objects accessed through an object behind the membrane are also behind the membrane. Common use cases of membranes are revoking all references to an object network at once or enforcing write protection on the objects behind the membrane [39, 26].

Figure 3 shows a membrane for *TargetA* implemented by wrapper *ProxyA*. Each property access through a wrapper (e.g., *ProxyA.x*) returns a wrapped object. After installing the membrane, no *new* direct references to target objects behind the membrane become available.

An *identity preserving membrane* guarantees that no target object has more than one proxy. Thus, proxy identity outside the membrane reflects target object identity inside. For example, if *TargetA.x.z* and *TargetA.y* refer to the same object (*TargetA.x.z===TargetA.y*),

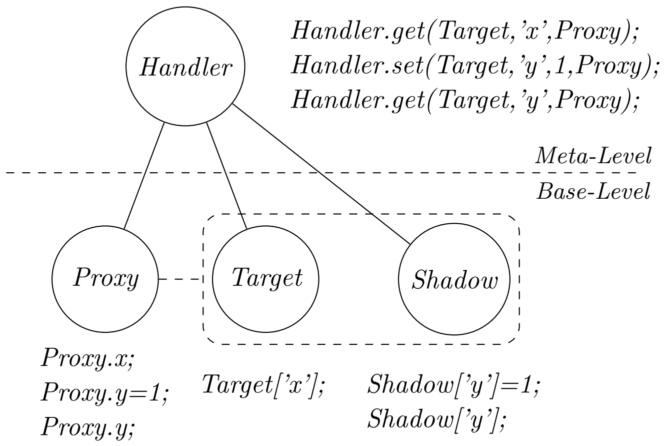


Figure 4 Operations on a sandbox. The property get operation *Proxy.x* invokes the trap *Handler.get(Target,'x',Proxy)*, which forwards the operation to the proxy's target. The property set operation *Proxy.y=1* invokes the trap *Handler.set(Target,'y',1,Proxy)*, which forwards the operation to a local shadow object. The final property get operation *Proxy.y* is than also forwarded to the shadow object.

then *ProxyA.x.z* and *ProxyA.y* refer to the same wrapper object (*ProxyA.x.z===ProxyA.y*).

3.2 Shadow objects

Our sandbox redefines the semantics of proxies to implement expanders [42], an idea that allows a client side extension of properties without modifying the proxy's target.

A sandbox handler manages two objects: a target object and a local *shadow object*. The target object acts as a parent object for its proxy whereas the shadow object gathers local modifications. Write operations always take place on the shadow object. A read operation first attempts to obtain the property from the shadow object. If that fails, the read gets forwarded to the target object. Figure 4 illustrates this behavior, which is very similar to JavaScript's prototype chain: the sandboxed version of an object inherits everything from its outside cousin, whereas modifications only appear inside the sandbox[3].

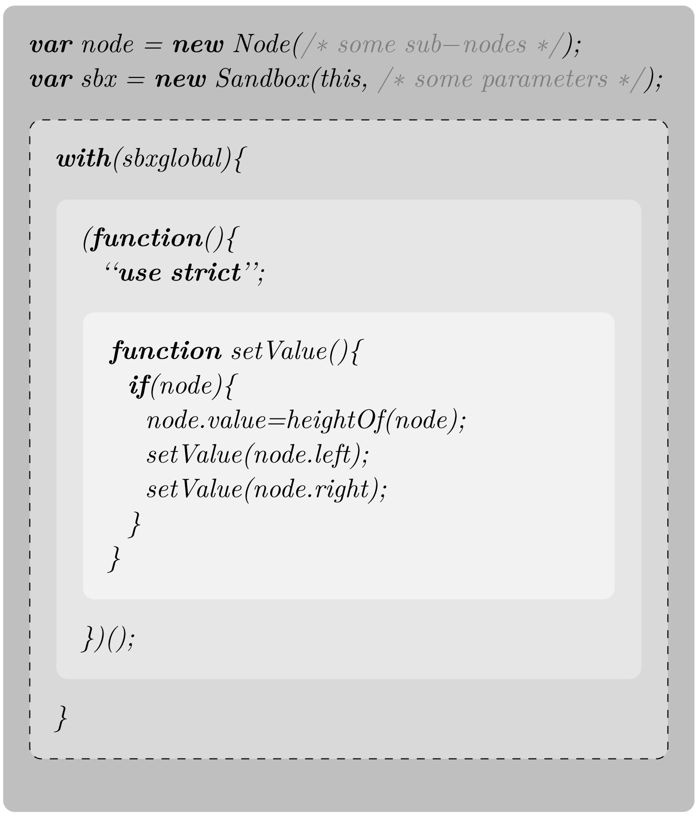
As sandbox encapsulation extends the functionality of a membrane, each object visible inside the sandbox is either an object that was created inside or it is a wrapper for some outside object.

A special proxy wraps sandbox internal values whenever committing a value to the outside, as shown in the last example. This step mediates uses of a sandbox internal value in the outside. This is form example required to wrap arguments values passed to committed sandbox function. The wrapping guarantees that the sandbox never gets access to unprotected references to the outside.

3.3 Sandbox scope

Apart from access restrictions, protecting the global state from modification through the membrane is fundamental to guarantee noninterference. To execute program code, DecentJS relies on an ***eval***, which is nested in a statement ***with*** *(sbxglobal) {/* body */}*. The ***with***

[3] Getter and setter functions require special treatment. Like other functions, they are decompiled and then applied to the shadow object. See Section 3.4.



```
var node = new Node(/* some sub—nodes */);
var sbx = new Sandbox(this, /* some parameters */);

with(sbxglobal){

  (function(){
    "use strict";

    function setValue(){
      if(node){
        node.value=heightOf(node);
        setValue(node.left);
        setValue(node.right);
      }
    }

  })();

}
```

Figure 5 Scope chain installed by the sandbox when loading *setValue*. The dark box represents the global scope. The dashed line indicates the sandbox boundary and the inner box shows the program code nested inside.

statement places the sandbox global on top of the current environment's scope chain while executing *body*. This setup exploits that ***eval*** dynamically rebinds the free variables of its argument to whatever is in scope at its call site. In this construction, which is related to dynamic binding [14], any property defined in *sbxglobal* shadows a variable deeper down in the scope chain.

We employ a proxy object in place of *sbxglobal* to control all non-local variable accesses in the sandboxed code by trapping the sandbox global object's *hasOwnProperty* method. When JavaScript traverses the scope chain to resolve a variable access, it calls the method *hasOwnProperty* on the objects of the scope chain starting from the top. Inside the ***with*** statement, the first object that is checked on this traversal is the proxied sandbox global. If its *hasOwnProperty* method always returns *true*, then the traversal stops here and the JavaScript engine sends all read and write operations for free variables to the sandbox global. This way, we obtain full interposition and the handler of the proxied sandbox global has complete control over the free variables in *body*.

Figure 5 visualizes the nested scopes created during the execution of *setValue* as in the example from Section 2. The sandbox global *sbxglobal* is a wrapper for the actual global object, which is used to access *heightOf* and *Math.abs*. The library code is nested in an empty closure which provides a fresh scope for local functions and variables. This step is required because JavaScript did not have standalone block scopes such as blocks in C or

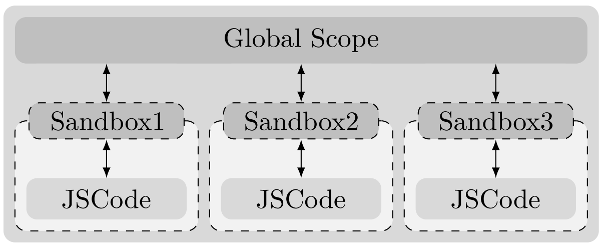


Figure 6 Nested sandboxes in an application. The outer box represents the global application state containing JavaScript's global scope. Each sandbox has its own global object and the nested JavaScript code is defined w.r.t. to the sandbox global.

Java. Variables and named functions [4] created by the sandboxed code end up in this fresh scope. This extra scope guarantees noninterference for dynamically loaded scripts that define global variables and functions.

The *"**use strict**"*[5] declaration in front of the closure puts JavaScript in strict mode, which ensures that the code cannot obtain unprotected references to the global object.

Figure 6 shows the situation when instantiating different sandboxes during program execution. Every sandbox installs its own scope with a sandbox global on top of the scope chain. Scripts nested inside are defined with respect to the sandbox global. The sandbox global mediates the access to JavaScript's global object. Its default implementation is empty to guarantee isolation. However, DecentJS can grant fine-grained access by making resources available in the sandbox global.

3.4 Function recompilation

In JavaScript, functions have access to the variables and functions in the lexical scope in which the function was defined. The Mozilla documentation[6] says: "It remembers the environment in which it was created.". Calls to wrapped functions may still cause side effects through their free variables (e.g., by modifying a variable or by calling another side-effecting function). Thus, sandboxing either has to erase external bindings of functions or it has to verify that a function is free of side effects. The former alternative is the default in DecentJS.

To remove bindings from functions passed through the membrane our protection mechanism decompiles the function and recompiles it inside the sandbox environment. Decompilation relies on the standard implementation of the *toString* method of a JavaScript function that returns a string containing the source code of the function. Each use of an external function in a sandbox first decompiles it by calling its *toString* method. To bypass potential tampering, we use a private copy of *Function.prototype.toString* for this call.

Next, we apply ***eval*** to the resulting string to create a fresh variant of the function. As explained in Section 3.3, this application of ***eval*** is nested in a ***with*** statement that supplies the desired environment. Decompilation also places a *"**use strict**"* statement in front.

To avoid a frequent decompilation and call of ***eval*** with respect to the same code, our implementation caches the compiled function where applicable.

[4] Function created with ***function*** *name() {/* body */}*.

[5] Strict mode requires that a use of *this* inside a function is only valid if either the function was called as a method or a receiver object was specified explicitly using *apply* or *call*.

[6] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures`

Instead of recompiling a function, we may use the string representation of a function to verify that a function is free of side effects, for example, by checking if the function's body is SES-compliant[7] [34]. However, it turns out that recompiling a function has a lower impact on the execution time than analyzing the function body.

Functions without a string representation (e.g., native functions like *Object* or *Array*) cannot be verified or sanitized before passing them through the membrane. We can either trust these functions or rule them out. To this end, DecentJS may be provided with a white list of trusted function objects. In any case, functions remain wrapped in a sandbox proxy to mediate property access.

In addition to normal function and method calls, the access to a property that is bound to a *getter* or *setter* function needs to decompile or verify the *getter* or *setter* before its execution.

3.5 DOM updates

The *Document Object Model* (DOM) is an API for manipulating HTML and XML documents that underlie the rendering of a web page. DOM provides a representation of the document's content and it offers methods for changing its structure, style, content, etc. In JavaScript, this API is implemented using special objects, reachable from the *document* object. Unfortunately, the document tree itself is not an object in the programming language. Thus, it cannot be wrapped for use inside of a sandbox. The only possibility is to wrap the interfaces, in particular, the *document* object.

We grant access to the DOM by binding the DOM interfaces to the sandbox global when instantiating a new sandbox. As all interfaces are wrapped in a sandbox proxy to mediate access, there are a number of limitations:

- By default, DOM nodes are accessed by calling query methods like *getElementById* on the *document* object. Effect logging recognizes these accesses as method calls, rather than as operations on the DOM.
- All query functions are special native functions that do not have a string representation. Decompilation is not possible so that using a query function must be permitted explicitly through the white list.
- A query function must be called as a method of an actual DOM object implementing the corresponding interface. Thus, DOM objects cannot be wrapped like other objects, but they require a special wrapping that calls the method on the correct receiver object. While read operations can be managed in this way, write operations must either be forbidden or they affect the original DOM.

Thus, guest code can modify the original DOM unless the DOM interface is restricted to read-only operations. With unrestricted operations it would be possible to insert new *<**script**>* elements in the document, which loads scripts from the internet and executes them in the normal application state without further sandboxing. However, prohibiting write operation means that the majority of guest codes cannot be executed in the sandbox.

To overcome this limitation, DecentJS provides guest code access to an *emulated* DOM instead of the real one. We rely on *dom.js*[8], a JavaScript library emulating a full browser DOM, to implement a DOM interface for scripts running in the sandbox. This emulated DOM is merged into the global sandbox object when executing scripts.

[7] In SES, a function can only cause side effects on values passed as an argument.
[8] `https://github.com/andreasgal/dom.js/`

As this pseudo DOM is constructed inside the sandbox, it can be accessed and modified at will. No special treatment is required. However, the pseudo DOM is wrapped in a special membrane mediating all operations and performing effect logging on all DOM elements.

As each sandbox owns a direct reference to the sandbox internal DOM it provides the following features to the user:

- The sandbox provides an interface to the sandbox internal DOM and enables the host program to access all aspects of the DOM. This interface can control the data visible to the guest program.
- A host can load a page template before evaluating guest code. This template can be an arbitrary HTML document, like the host's page or a blank web page. As most libraries operate on non-blank page documents (e.g., by reading or writing to a particular element) this template can be used to create an environment.
- Guest code runs without restrictions. For example, guest code can introduce new <***script***> elements to load library code from the internet. These libraries are loaded and executed inside the sandbox as well.
- All operations on the interface objects are recorded, for example, the access to *window .location* when loading a document. Effects can examined using a suitable API (cf. Section 2.2).
- The host program can perform a fine-grained inspection of the document tree (e.g., it can search for changes and differences). The host recognizes newly created DOM elements and it can transfer content from the sandbox DOM to the DOM of the host program.

3.6 Policies

A policy is a guideline that prescribes whether an operation is allowed. Most existing sandbox systems come with a facility to define policies. For example, a policy may grant access to a certain resource, it may grant the right to perform an operation or to cause a side effect.

Our system does not provide access control policies in the manner known from other systems. DecentJS only provides the mechanism to implement an empty scope and to pass selected resources to this scope. When a reference to a certain resource is made available inside the sandbox, then it should be wrapped in another proxy membrane that enforces a suitable policy.

For example, one may use this work's transactional membranes to shadow write operations, *Access Permission Contracts* [18] to restrict the access on objects, or *Revocable References* [39, 40] to revoke access to the outside world.

4 Discussion

Strict Mode

DecentJS runs guest code in JavaScript's strict mode to rule out uncontrolled accesses to the global object. This restriction may lead to dysfunctional guest code because strict-mode semantics is subtly different from non-strict mode JavaScript.

However, assuming strict mode is less restrictive than the restrictions imposed by other techniques that restrict JavaScript's dynamic features. Alternatively, one could also provide a program transformation that guards uses of *this* that may access the global object.

Scopes

DecentJS places every load in its own scope. Hence, variables and functions declared in one script are not visible to the execution of another script in the same sandbox. Indeed, we deliberately keep scopes apart to avoid interference. To enable communication DecentJS offers a facility to load mutually dependant scripts into the same scope. Otherwise, scripts may exchange data by writing to fields in the sandbox global object.

Function Decompilation

If a top level closure is wrapped in a sandbox, then its free valriables have to be declared to the sandbox or their bindings are removed. Decompilation may change the meaning of a function, because it rebinds its free variables. Only “pure functions”[9] can be decompiled without changing their meaning.

However, decompiling preserves the semantics of a function if its free variables are imported in the sandbox. The new closure formed within the sandbox may be closed over variables defined in that sandbox. This task is rightfully manual as the availability of global bindings is part of a policy.

In conclusion, decompilation is unavoidable to guarantee noninterference of a function defined in another scope as every property read operation may be the call of a side-effecting getter function.

Native Functions

Decompilation requires a string that contains the source code of that function, but calling the standard *toString* method from *Function.prototype* does not work for all functions.

- A native function does not have a string representation. Trust in a native function is regulated with a white list of trusted functions.
- The *Function.prototype.bind()* method creates a new function with the same body, but the first couple of arguments bound to the arguments of *bind()*. JavaScript does not provide a string representation for the newly created function.

Object, Array, and Function Initializer

In JavaScript, some objects can be initialized using a *literal notation* (initializer notation). Examples are object literals (using *{}*), array objects (using *[]*), and function objects (using the named or unnamed function expression, e.g. ***function*** *(){}*). Using the literal notation circumvents all restrictions and wrappings that we may have imposed on the *Object*, *Array*, and *Function* constructors.

As we are not able to intercept the construction using the literal notation enables unprotected read access to the prototype objects *Object.prototype*, *Array.prototype*, and *Function.prototype*. The newly created object always inherits from the corresponding prototype.

However, we will never get access to the prototype object itself and we are not able to modify the prototype. Writes to the created objects always effect the object itself and are never forwarded to the prototype object.

[9] A pure function is a function that only maps its input into an output without causing any observable side effect.

Even though all the elements contained in the native prototype objects are uncritical by default, a global (not sandboxed) script could add sensitive data or a side effecting function to one of the prototype objects and thus bypass access to unprotected data.

Function Constructor

The function constructor *Function* creates a new function object based on the definition given as arguments. In contrast to function statements and function expressions, the function constructor ignores the surrounding scope. The new function is always created in the global scope and calling it enables access to all global variables.

To prevent this leakage, the sandbox never grants unwrapped access to JavaScipt's global *Function* constructor even if the constructor is white-listed as a safe native function. A special wrapping intercepts the operations and uses a safe way to construct a new function with respect to the sandbox.

Noninterference

The execution of sandboxed code should not interfere with the execution of application code. That is, the application should run as if no sandboxes were present. This property is called *noninterference* (NI) [10] by the security community. The intuition is that sandboxed code runs at a lower level of security than application code and that the low-security sandbox code must not be able to observe the results of the high-security computation in the global scope.

DecentJS guarantees integrity and confidentiality. The default "empty" sandbox guarantees to run code in full isolation from the rest of the application, whereas the sandbox global can provide protected references to the sandbox.

In summary, the sandboxed code may try to write to an object that is visible to the application, it may throw an exception, or it may not terminate. Our membrane redirects all write operations in sandboxed code to local replicas and it captures all exceptions. A timeout could be used to transform non-terminating executions into an exception, alas such a timeout cannot be implemented in JavaScript.[10]

5 Evaluation

To evaluate our implementation, we applied it to JavaScript benchmark programs from the Google Octane 2.0 Benchmark Suite[11]. These benchmarks measure a JavaScript engine's performance by running a selection of complex and demanding programs (benchmark programs run between 5 and 8200 times). Google claims that Octane "measure[s] the performance of JavaScript code found in large, real-world web applications, running on modern mobile and desktop browsers. Each benchmark is complex and demanding .

As expected, the run time increases when executing a benchmark in a sandbox. While some programs like *EarleyBoyer*, *NavierStrokes*, *pdf.js*, *Mandreel*, and *Box2DWeb* are heavily affected, others are only slightly affected: *Richards*, *Crypto*, *RegExp*, and *Code loading*, for instance. The observed run time impact entirely depends on the number of values that cross the membrane.

[10] The JavaScript *timeout* function only schedules a function to run when the currently running JavaScript code—presumably some event handler—stops. It cannot interrupt a running function.
[11] `https://developers.google.com/octane`

From the running times we find that the sandbox itself causes an average slowdown of 8.01 (over all benchmarks). This is more than acceptable compared to other language-embedded systems. The numbers also show that sandboxing with fine-grained effect logging enabled causes an average slowdown of 32.60, an additional factor of 4.07 on top of pure sandboxing.

Because the execution of program code inside of a sandbox is nothing else than a normal program execution inside of a ***with*** statement and with one additional call to ***eval*** (when instantiating the execution) the run-time impact is influenced by (i) the number of wrap operations of values that cross the membrane, (ii) the number of decompile operations on functions, and (iii) the number of effects on wrapped objects. Readouts from internal counters indicate that the heavily affected benchmarks (*RayTrace*, *pdf.js*, *Mandreel*, and *Box2DWeb*) perform a very large number of effects. The *RayTrace* benchmark, for example, performs 51 million effects.

Overall, an average slowdown of 8.01 is more than acceptable compared to other language-embedded systems. As Octane is intended to measure the engine's performance (benchmark programs run between 5 and 8200 times) we claim that it is the heaviest kind of benchmark. Every real-world library (e.g. *jQuery*) is less demanding and runs without an measurable runtime impact.

Appendix F also contains the score values obtained from running the benchmark suite and lists the readouts of some internal counters.

6 Conclusion

DecentJS runs JavaScript code in a configurable degree of isolation with fine-grained access control rather than disallowing all access to the application state. It provides full browser compatibility (i.e. all browsers work without modifications as long as the proxy API is supported) and it has a better performance than other language-embedded systems.

Additionally, DecentJS comes with the following features:

1. *Language-embedded sandbox.* DecentJS is a JavaScript library and all aspects are accessible through a sandbox API. The library can be deployed as a language extension and requires no changes in the JavaScript run-time system.
2. *Full interposition.* DecentJS is implemented using JavaScript proxies [39]. The proxy-based implementation guarantees full interposition for the full JavaScript language including all dynamic features (e.g., ***with***, ***eval***). DecentJS works for all code regardless of its origin, including dynamically loaded code and code injected via ***eval***. No source code transformation or avoidance of JavaScript's dynamic features is required.
3. *Transaction-based sandboxing.* A DecentJS sandbox provides a transactional scope that logs all effects. Wrapper proxies make external objects accessible inside of the sandbox and enable sandbox internal modifications of the object. Hence, sandboxed code runs as usual without noticing the sandbox. Effects reveal conflicts, differences, and changes with respect to another sandbox or the global state. After inspection of the log, effects can be committed to the application state or rolled back.

Acknowledgments This work benefited from discussions with participants of the Dagstuhl Seminar "Scripting Languages and Frameworks: Analysis and Verification" in 2014. In particular, Tom Van Cutsem provided helpful advice on the internals of JavaScript proxies.

References

1 Adsafe: Making JavaScript safe for advertising. `http://www.adsafe.org/`, 2015.

2 P. Agten, S. V. Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In R. H. Zakon, editor, *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 1–10. ACM, 2012.

3 J. Arnaud, M. Denker, S. Ducasse, D. Pollet, A. Bergel, and M. Suen. Read-only execution for dynamic languages. In J. Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 117–136, Málaga, Spain, June 2010. Springer.

4 A. Charguéraud. Pretty-big-step semantics. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60, Rome, Italy, Mar. 2013. Springer.

5 A. Dewald, T. Holz, and F. C. Freiling. ADSandbox: sandboxing JavaScript to fight malicious websites. In S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 1859–1864, Sierre, Switzerland, 2010. ACM.

6 M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, pages 382–391. IEEE Computer Society, 2009.

7 M. Dhawan, C. Shan, and V. Ganapathy. Enhancing JavaScript with transactions. In J. Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 383–408. Springer, 2012.

8 Facebook SDK for JavaScript. `https://developers.facebook.com/docs/javascript/`, 2015.

9 A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to strangers without taking their candy: isolating proxied content. In L. Stein and A. Mislove, editors, *Proceedings of the 1st Workshop on Social Network Systems, SNS 2008, Glasgow, Scotland, UK, April 1, 2008*, pages 25–30. ACM, 2008.

10 J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

11 google-caja: A source-to-source translator for securing JavaScript-based web content. `http://code.google.com/p/google-caja/`, (as of 2011).

12 S. Guarnieri and V. B. Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In F. Monrose, editor, *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 151–168. USENIX Association, 2009.

13 A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In T. D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 126–150, Maribor, Slovenia, June 2010. Springer.

14 D. R. Hanson and T. A. Proebsting. Dynamic variables. In *Proceedings of the 2001 Conference on Programming Language Design and Implementation*, pages 264–273, Snowbird, UT, USA, June 2001. ACM Press, New York, USA.

15 D. Hedin et al. JSFlow: Tracking information flow in JavaScript and its APIs. In *ACM Symposium on Applied Computing (SAC'14)*, Gyeongju, Korea, Mar. 2014.

16 P. Heidegger, A. Bieniusa, and P. Thiemann. Access permission contracts for scripting languages. In J. Field and M. Hicks, editors, *Proceedings 39th Annual ACM Symposium on*

Principles of Programming Languages, pages 111–122, Philadelphia, USA, Jan. 2012. ACM Press.

17 P. Heidegger and P. Thiemann. A heuristic approach for computing effects. In J. Bishop and A. Vallecillo, editors, *Proceedings of the 49th International Conference on Objects, Models, Components and Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 147–162, Zurich, Switzerland, June 2011. Springer.

18 M. Keil and P. Thiemann. Efficient dynamic access analysis using JavaScript proxies. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 49–60, New York, NY, USA, 2013. ACM.

19 M. Keil and P. Thiemann. Efficient dynamic access analysis using JavaScript proxies. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 49–60, Indianapolis, Indiana, USA, 2013. ACM.

20 M. Keil and P. Thiemann. Treatjs: Higher-order contracts for JavaScript. In J. Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPICS*, pages 28–51, Prague, Czech Repulic, July 2015. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

21 M. Keil and P. Thiemann. TreatJS: Higher-order contracts for JavaScript. Technical report, Institute for Computer Science, University of Freiburg, 2015.

22 M. Keil and P. Thiemann. TreatJS Online. `http://www2.informatik.uni-freiburg.de/~keilr/treatjs/`, 2015.

23 S. Maffeis and A. Taly. Language-based isolation of untrusted JavaScript. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 77–91. IEEE Computer Society, 2009.

24 J. Magazinius, P. H. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In T. Aura, editor, *The 15th Nordic Conference in Secure IT Systems*, Lecture Notes in Computer Science. Springer Verlag, Oct. 2010.

25 L. A. Meyerovich and V. B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy*, pages 481–496, Berkeley/Oakland, California, USA, May 2010. IEEE Computer Society.

26 M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006. AAI3245526.

27 M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. `http://google-caja.googlecode.com`, 2008. Google White Paper.

28 M. S. Miller and J. S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In V. A. Saraswat, editor, *Advances in Computing Science - ASIAN 2003 Programming Languages and Distributed Computation, 8th Asian Computing Science Conference, Mumbai, India, December 10-14, 2003, Proceedings*, volume 2896 of *Lecture Notes in Computer Science*, pages 224–242. Springer, 2003.

29 K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang. Towards fine-grained access control in JavaScript contexts. In *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*, pages 720–729. IEEE Computer Society, 2011.

30 P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In W. Li, W. Susilo, U. K. Tupakula, R. Safavi-Naini, and V. Varadharajan, editors, *ASIACCS*, pages 47–60, Sydney, Australia, Mar. 2009. ACM.

31 J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

32 G. Richards, C. Hammer, F. Z. Nardelli, S. Jagannathan, and J. Vitek. Flexible access control for JavaScript. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 305–322. ACM, 2013.

33 Same-origin policy. `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy/`, 2008.

34 SecureEcmaScript (ses). `https://code.google.com/p/es-lab/wiki/SecureEcmaScript`, 2015.

35 N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles of Distributed Computing*, pages 204–213, Ottowa, Ontario, Canada, 1995. ACM Press, New York, NY, USA.

36 Signed scripts in Mozilla. `http://www-archive.mozilla.org/projects/security/components/signed-scripts.html/`, 2015.

37 T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In G. T. Leavens and M. B. Dwyer, editors, *OOPSLA*, pages 943–962. ACM, 2012.

38 J. Terrace, S. R. Beard, and N. P. K. Katta. JavaScript in JavaScript (js.js): Sandboxing third-party scripts. In E. M. Maximilien, editor, *3rd USENIX Conference on Web Application Development, WebApps'12, Boston, MA, USA, June 13, 2012*, pages 95–100. USENIX Association, 2012.

39 T. Van Cutsem and M. S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In W. D. Clinger, editor, *DLS*, pages 59–72. ACM, 2010.

40 T. Van Cutsem and M. S. Miller. Trustworthy proxies - virtualizing objects with invariants. In G. Castagna, editor, *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 154–178, Montpellier, France, July 2013. Springer.

41 G. Wagner, A. Gal, C. Wimmer, B. Eich, and M. Franz. Compartmental memory management in a modern web browser. In H. Boehm and D. F. Bacon, editors, *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, pages 119–128. ACM, 2011.

42 A. Warth, M. Stanojevic, and T. Millstein. Statically scoped object adaptation with expanders. In *Proceedings of the 21th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 37–56, Portland, OR, USA, 2006. ACM Press, New York.

43 G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

```html
<!DOCTYPE html>
  <html lang="en">
  <head>
    <!-- third-party libraries -->
    <script src="date.js"></script>
    <script src="jquery.js"></script>
    <script src="jquery.formatDateTime.js"></script>
  </head>
  <body>
    <!-- Body of the page -->
    <h1 id="headline">Headline</h1>
    <script type="text/javascript">
      window.$("#headline").text("Changed Headline");
    </script>
  </body>
</html>
```

Figure 7 Motivating Example. The listing shows a snippet of an *index.html* file. The <***script***> tags load third-party libraries to the application state before executing the body. Within the <***body***> tag it uses *jQuery* to modify the DOM.

A Motivation

JavaScript is the most important client side language for web pages. JavaScript developers rely heavily on third-party libraries for calenders, maps, social networking, feature extensions, and so on. Thus, the client-side code of a web page is usually composed of dynamically loaded fragments from different origins.

However, the JavaScript language has no provision for namespaces or encapsulation management: there is a global scope for variables and functions, and every loaded script has the same authority. On the one hand, JavaScript developers benefit from JavaScript's flexibility as it enables to extend the application state easily. On the other hand, once included, a script has the ability to access and manipulate every value reachable from the global object. That makes it difficult to enforce any security policy in JavaScript.

As a consequence, program understanding and maintenance becomes very difficult because side effects may cause unexpected behavior. There is also a number of security concerns as the library code may access sensitive data, for example, it may read user input from the browser's DOM.

Browsers normally provide build-in isolation mechanisms. However, as isolation is not always possible for all scrips, the key challenges of a JavaScript developer is to manage untrusted third-party code, to control the use of data by included scrips, and to reason about effects of included code.

```
<!DOCTYPE html>
  <html lang="en">
  <head>
    <!-- DecentJS code-->
    <script src="decent.js"></script>
    <!-- Runs Datejs in a fresh sandbox. -->
    <script type="text/javascript">
      var sbx = new Sandbox(this, Sandbox.DEFAULT);
      sbx.request("datejs.js");
      sbx.applyRule(
        new Rule.CommitOn(Date, function(sbx, effect) {
          return (effect instanceof Effect.Set) &&
            !(effect.name in Date);
      }));
    </script>
    <!-- ... -->
    </head>
  <body>
    <!-- ... -->
  </body>
</html>
```

Figure 8 Execution of library code in a sandbox. The first <***script***> tag loads the sandbox implementation. The body of the second <***script***> tag instantiates a new sandbox and loads and executes *Datejs* inside the sandbox. Later it commits intended effects to the native *Date* object.

A.1 JavaScript issues

As an example, we consider a web application that relies on third-party scripts from various sources. Figure 7 shows an extract of such a page. It first includes *Datejs*[12], a library extending JavaScript's native *Date* object with additional methods for parsing, formatting, and processing of dates. Next, it loads *jQuery*[13] and a *jQuery* plugin *jquery.formatDateTime.js*[14] that also simplifies formatting of JavaScript date objects.

At this point, we want to ensure that loading the third-party code (*Datejs* and *jQuery*) does not influence the application state in an unintended way. Encapsulating the library code in a sandbox enables us to scrutinize modifications that the foreign code may attempt and only commit acceptable modifications.

A.2 Isolating third-party JavaScript

Transactional sandboxing is inspired by the idea of transaction processing in database systems [43] and software transactional memory [35]. Each sandbox implements a transactional scope the content of which can be examined, committed, or rolled back.

1. *Isolation of code.* A DecentJS sandbox can run JavaScript code in isolation to the

[12] `https://github.com/datejs/Datejs`
[13] `https://jquery.com/`
[14] `https://github.com/agschwender/jquery.formatDateTime`

application state. Proxies make external values visible inside of the sandbox and handle sandbox internal write operations. An internal DOM simulates the browser DOM as needed. This setup guarantees that the isolated code runs without noticing the sandbox.

2. *Providing transactional features.* A DecentJS sandbox provides a transactional scope in which effects are logged for inspection. Policy rules can be specified so that only effects that adhere to the rules are committed to the application state and others are rolled back.

Appendix 2 gives a detailed introduction to DecentJS's API and provides a series of examples explaining its facilities.

Figure 8 shows how to modify the *index.html* from Figure 7 to load the third-party code into a sandbox. We first focus on *Datejs* and consider *jQuery* later in Section A.4. The *<!-- ... -->* comment is a placeholder for unmodified code not considered in this example[15].

Initially, we create a fresh sandbox (line 24). The first parameter is the sandbox-internal global object for scripts running in the sandbox whereas the second parameter is a configuration object[16].

The sandbox global object acts as a mediator between the sandbox contents and the external world (cf. Section 3.3). It is placed on top of the scope chain for code executing inside the sandbox and it can be used to make outside values available inside the sandbox. It is wrapped in a proxy membrane to mediate all accesses to the host program.

Next, we instruct the sandbox to load and execute the *Datejs* library (line 25) inside the sandbox. Afterwards, the sandbox-internal proxy for JavaScript's native *Date* object is modified in several ways. Among others, the library adds new methods to the *Date* object and extends *Date.prototype* with additional properties. Write operations on a proxy wrapper produce a *shadow value* (cf. Section 3.2) that represents the sandbox-internal modification of an object. Initially, this modification is only visible to reads inside the sandbox. Reads are forwarded to the target unless there are local modifications, in which case the shadow value is returned. The return values are wrapped in proxies, again.

A.3 Committing intended modifications

During execution, each sandbox records the effects on all objects that cross the sandbox membrane[17]. The sandbox API offers access to the resulting lists for inspection and provides query methods to select the effects of a particular object. After loading *Datejs*, the effect log reports 16 reads and 142 writes on three different objects[18]. However, as the manual inspection of effects is impractical and requires a lot of effort, DecentJS allows us to register *rules* with a sandbox and apply them automatically. A rule combines a sandbox operation with a predicate specifying the state under which the operation is allowed to be performed.

For example, as we consider an extension to the *Date* object as intended, non-critical modification, we install a rule that automatically commits new properties to the *Date* object in Line 26. In general, a rule *CommitOn* takes a target object (*Date*) and a predicate. The predicate function gets invoked with the *sandbox object* (*sbx*) and an *effect object* describing an effect on the target object. In our example, the predicate checks if the effect is a property

[15] Appendix A.6 shows the full HTML code.
[16] *Sandbox.DEFAULT* is a predefined configuration object for the standard use of the sandbox. It consists of simple key-value pairs, e.g. *verbose:false*.
[17] The lists do not contain effects on values that were created inside of the sandbox.
[18] Appendix A.7 shows a readout of the effect lists.

write operation extending JavaScript's native *Date* object and that the property name is not already present.

If we construct a function inside of a sandbox and this function is written and committed to an outside object, then the free variables of the function contain objects inside the sandbox and arguments of a call to this function are also wrapped. That is, calling this function on the outside only causes effects inside the sandbox. Furthermore, committing an object in this way wraps the object in a proxy before writing it to its (outside) target. Both measures are required to guarantee that the sandbox never gets access to unwrapped references from the outside world.

At this point we have to mention that the data structure of the committed functions is constructed inside of *sbx*. All bound references of those functions still point to objects inside of the sandbox and thus using them only causes effect inside of the sandbox. Furthermore, committing an object wraps the object in a proxy before writing it to its target. This intercepts the use of the committed object, e.g. to wrap the arguments of a committed function before invoking the function. This is

As an illustration, Figure 9 shows an extract of the membrane arising from JavaScript's native *Date* object in Appendix **??**. Executing *Datejs* in *sbx* (shown on the left in the first box) creates a proxy for each element accessed on *Date*: *Date* and *Date.prototype*. Only *Date* and *Date.prototype* are wrapped because proxies are created on demand. As proxies forward each read to the target the structure visible inside of the sandbox is identical to the structure visible outside.

Extending the native *Date* object in *sbx* yields the state shown in the second box. All modifications are only visible inside of the sandbox. The new elements are not wrapped because they only exist inside of the sandbox.

However, a special proxy wraps sandbox internal values whenever committing a value to the outside, as shown in the last box. This step mediates further uses of the sandbox internal value, for example, wrapping the *this* value and all arguments when calling a function defined in the sandbox. The wrapping guarantees that the sandbox never gets access to unprotected references to the outside.

A.4 Shadowing DOM operations

The example in Figure 8 omits the inclusion of *jQuery* for simplification purposes. However, our initial objective is to sandbox all third-party code to i reason about the modifications done by loading the third-party code ii prevent the application state from unintended modifications .

Isolating a library like *jQuery* is more challenging as it needs access to the browser's DOM. Calls to the native DOM interface expose a mixture of public and confidential information, so the access can neither be fully trusted nor completely forbidden. To address this issue, DecentJS provides an *internal DOM*[19] that serves as a shadow for the actual DOM when running a web library in the sandbox.

Figure 10 demonstrates loading the *jQuery* library in a web sandbox. As we extend the first example, we create a new empty sandbox (line 46) and initialize the sandbox internal DOM by loading an HTML template (line 47). Using the *Sandbox.WEB* configuration activates the shadow DOM by instructing the sandbox to create a DOM interface and to merge this interface with the sandbox-internal global object. The shadow DOM initially

[19] See Section 3.5 for a more detailed discussion.

contains an empty document. It can be instantiated with the actual HTML body or with an HTML template, as shown in Line 47.

Figure 11 shows the template, which is an extract of the original *index.html* containing only the *<script>* tags for the *jQuery* library and selected parts of the HTML body. Loading the template also loads and executes the third-party code inside the sandbox. Afterwards, the internal effect log reports two write operations to the fields *$* and *jQuery* of the global *window* object, and one write operation to the *HTMLBodyElement* interface, a child of *Node*, both of which are part of the DOM interface.

To automatically commit intended modifications to the global *window* object, we install a suitable rule in Lines 48 and 49. As *jQuery* has been instantiated w.r.t. the sandbox internal DOM, using it modifies the sandbox internal DOM instead of the browser's DOM. These modifications must be committed to the browser's DOM to become visible (line 57).

Alternatively, DecentJS allows us to grant access to the browser's DOM by white listing the *window* and *document* objects. However, white listing can only expose entire objects and cannot restrict access to certain parts of the document model.

A.5 Using transactions

For wrapped objects, DecentJS supports a commit/rollback mechanism. In the first examples (Figure 8), we prevent the application state from unintended modification when loading untrusted code and commit only intended ones.

However, *Datejs* and *jquery.formatDateTime.js* might both modify JavaScript's native *Date* object. To avoid undesired overwrites, DecentJS allows us to inspect the effects of both libraries and to check for conflicts before committing to *Date*. The predicate in Line 81 checks for *conflicts*, which arise in the comparison between different sandboxes. A conflict is flagged if at least one sandbox modifies a value that is accessed by the other sandbox later on[20].

Furthermore, we prescribe that in case of conflicts the methods from *Datejs* should be used. To this end, a second rule discards the modifications on *Date* from the second sandbox when detecting conflicts. The *rollback* operation undoes an existing manipulation and returns to its previous configuration. Such a partial rollback does not result in an inconsistent state as we do not delete objects and the references inside the sandbox remain unchanged.

A.6 Full HTML Example

Figure 13 shows the full html code from the example in Section A.

A.7 Effects Lists

This sections shows the resulting effect logs recorded by the sandboxes in Section A. See Appendix **??** for a detailed explanation of the output.

A.8 Effects of *sbx*

A.8.1 All Read Effects on *this*

[20] We consider only *Read-After-Write* and *Write-After-Write* conflicts. *Write-after-Read* conflicts are not handled because the hazard represents a problem that only occurs in concurrent executions.

```
sbx.readeffectOn(this).forEach(function(e) {
  print(e);
});
```

```
(#0) has [name=Date]
(#0) get [name=Date]
(#0) has [name=Number]
(#0) get [name=Number]
(#0) has [name=RegExp]
(#0) get [name=RegExp]
(#0) has [name=Array]
(#0) get [name=Array]
```

A.8.2 All Write Effects on *this*

```
sbx.writeeffectOn(this).forEach(function(e) {
  print(e);
});
```

```
none
```

A.8.3 All Read Effects on *Date*

```
sbx.readeffectOn(Date).forEach(function(e) {
  print(e);
});
```

```
(#1) get [name=prototype]
(#1) get [name=Parsing]
(#1) get [name=Grammar]
(#1) get [name=Translator]
(#1) get [name=CultureInfo]
(#1) get [name=parse]
```

A.8.4 All Write Effects on *Date*

```
sbx.writeeffectOn(Date).forEach(function(e) {
  print(e);
});
```

```
(#1) set [name=CultureInfo]
(#1) set [name=getMonthNumberFromName]
(#1) set [name=getDayNumberFromName]
(#1) set [name=isLeapYear]
(#1) set [name=getDaysInMonth]
(#1) set [name=getTimezoneOffset]
(#1) set [name=getTimezoneAbbreviation]
(#1) set [name=_validate]
(#1) set [name=validateMillisecond]
(#1) set [name=validateSecond]
(#1) set [name=validateMinute]
```

```
(#1) set [name=validateHour]
(#1) set [name=validateDay]
(#1) set [name=validateMonth]
(#1) set [name=validateYear]
(#1) set [name=now]
(#1) set [name=today]
(#1) set [name=Parsing]
(#1) set [name=Grammar]
(#1) set [name=Translator]
(#1) set [name=_parse]
(#1) set [name=parse]
(#1) set [name=getParseFunction]
(#1) set [name=parseExact]
```

A.8.5 All Read Effects on *Date.prototype*

```
sbx.readeffectOn(Date.prototype).forEach(function(e) {
  print(e);
});
```

```
(#2) get [name=toString]
```

A.8.6 All Write Effects on *Date.prototype*

```
sbx.writeeffectOn(Date.prototype).forEach(function(e) {
  print(e);
});
```

```
(#2) set [name=clone]
(#2) set [name=compareTo]
(#2) set [name=equals]
(#2) set [name=between]
(#2) set [name=addMilliseconds]
(#2) set [name=addSeconds]
(#2) set [name=addMinutes]
(#2) set [name=addHours]
(#2) set [name=addDays]
(#2) set [name=addWeeks]
(#2) set [name=addMonths]
(#2) set [name=addYears]
(#2) set [name=add]
(#2) set [name=set]
(#2) set [name=clearTime]
(#2) set [name=isLeapYear]
(#2) set [name=isWeekday]
(#2) set [name=getDaysInMonth]
(#2) set [name=moveToFirstDayOfMonth]
(#2) set [name=moveToLastDayOfMonth]
(#2) set [name=moveToDayOfWeek]
(#2) set [name=moveToMonth]
```

```
(#2) set [name=getDayOfYear]
(#2) set [name=getWeekOfYear]
(#2) set [name=isDST]
(#2) set [name=getTimezone]
(#2) set [name=setTimezoneOffset]
(#2) set [name=setTimezone]
(#2) set [name=getUTCOffset]
(#2) set [name=getDayName]
(#2) set [name=getMonthName]
(#2) set [name=_toString]
(#2) set [name=toString]
(#2) set [name=_orient]
(#2) set [name=next]
(#2) set [name=previous]
(#2) set [name=prev]
(#2) set [name=last]
(#2) set [name=_is]
(#2) set [name=is]
(#2) set [name=sun]
(#2) set [name=sunday]
(#2) set [name=mon]
(#2) set [name=monday]
(#2) set [name=tue]
(#2) set [name=tuesday]
(#2) set [name=wed]
(#2) set [name=wednesday]
(#2) set [name=thu]
(#2) set [name=thursday]
(#2) set [name=fri]
(#2) set [name=friday]
(#2) set [name=sat]
(#2) set [name=saturday]
(#2) set [name=jan]
(#2) set [name=january]
(#2) set [name=feb]
(#2) set [name=february]
(#2) set [name=mar]
(#2) set [name=march]
(#2) set [name=apr]
(#2) set [name=april]
(#2) set [name=may]
(#2) set [name=jun]
(#2) set [name=june]
(#2) set [name=jul]
(#2) set [name=july]
(#2) set [name=aug]
(#2) set [name=august]
(#2) set [name=sep]
```

```
(#2) set [name=september]
(#2) set [name=oct]
(#2) set [name=october]
(#2) set [name=nov]
(#2) set [name=november]
(#2) set [name=dec]
(#2) set [name=december]
(#2) set [name=milliseconds]
(#2) set [name=millisecond]
(#2) set [name=seconds]
(#2) set [name=second]
(#2) set [name=minutes]
(#2) set [name=minute]
(#2) set [name=hours]
(#2) set [name=hour]
(#2) set [name=days]
(#2) set [name=day]
(#2) set [name=weeks]
(#2) set [name=week]
(#2) set [name=months]
(#2) set [name=month]
(#2) set [name=years]
(#2) set [name=year]
(#2) set [name=toJSONString]
(#2) set [name=toShortDateString]
(#2) set [name=toLongDateString]
(#2) set [name=toShortTimeString]
(#2) set [name=toLongTimeString]
(#2) set [name=getOrdinal]
```

A.9 Effects of *sbx2*

A.9.1 All Read Effects on *this*

```
sbx2.readeffectOn(this).forEach(function(e) {
  print(e);
});
```

```
(#0) has [name=window]
(#0) get [name=window]
(#0) has [name=module]
(#0) get [name=module]
(#0) has [name=Math]
(#0) get [name=Math]
(#0) has [name=Array]
(#0) get [name=Array]
(#0) has [name=Date]
(#0) get [name=Date]
(#0) has [name=undefined]
(#0) get [name=undefined]
```

```
(#0) has [name=Symbol]
(#0) get [name=Symbol]
(#0) has [name=RegExp]
(#0) get [name=RegExp]
(#0) has [name=String]
(#0) get [name=String]
(#0) has [name=define]
(#0) get [name=define]
```

A.9.2 All Write Effects on *this*

```
sbxs.writeeffectOn(this).forEach(function(e) {
  print(e);
});
```

```
none
```

A.9.3 All Read Effects on *window*

```
sbx2.readeffectOn(window).forEach(function(e) {
  print(e);
});
```

```
(#1) get [name=window]
(#1) getOwnPropertyDescriptor [name=window]
(#1) getOwnPropertyDescriptor [name=module]
(#1) get [name=document]
(#1) getOwnPropertyDescriptor [name=Math]
(#1) getOwnPropertyDescriptor [name=Array]
(#1) getOwnPropertyDescriptor [name=Date]
(#1) getOwnPropertyDescriptor [name=undefined]
(#1) getOwnPropertyDescriptor [name=Symbol]
(#1) getOwnPropertyDescriptor [name=RegExp]
(#1) get [name=top]
(#1) get [name=setTimeout]
(#1) has [name=onfocusin]
(#1) get [name=location]
(#1) getOwnPropertyDescriptor [name=String]
(#1) get [name=XMLHttpRequest]
(#1) getOwnPropertyDescriptor [name=define]
(#1) get [name=jQuery]
(#1) get [name=$]
```

A.9.4 All Write Effects on *window*

```
sbx.writeeffectOn(window).forEach(function(e) {
  print(e);
});
```

```
(#1) set [name=$]
(#1) set [name=jQuery]
```

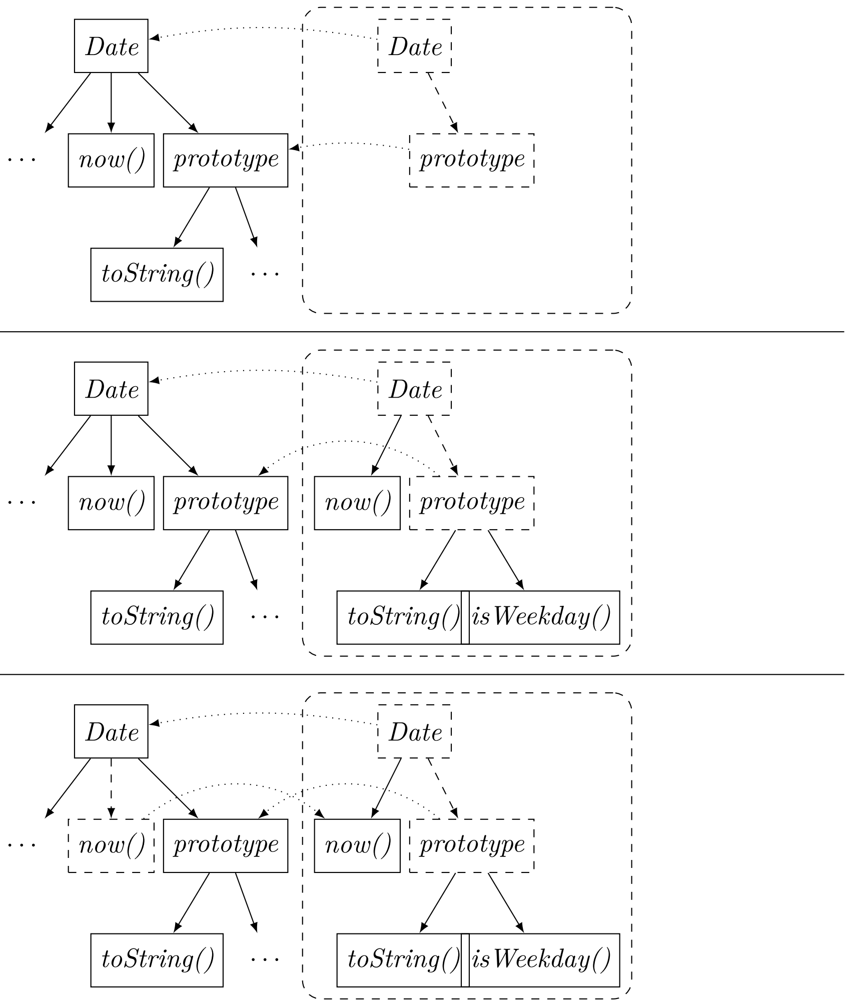


Figure 9 Shadow objects in the sandbox when loading *Datejs* (cf. Section **??**). The structure of JavaScrip's native *Date* object is shown in solid lines on the left. The shadow values are enclosed by a dashed line. Solid lines are direct references to non-proxy objects, whereas dashed lines are indirect references and proxy objects. Dotted lines connect to the target object. The first box shows the sandbox after reading *Date.prototype* whereas the second box shows the sandbox after modifying the structure of *Date*. The third box shows the situation after committing the modifications on *Date*.

```
<!DOCTYPE html>
  <html lang="en">
  <head>
    <!-- DecentJS code-->
    <script src="decent.js"></script>
    <!-- ... -->
    <!-- Runs jQuery in a fresh sandbox. -->
    <script type="text/javascript">
      var sbx2 = new Sandbox(this, Sandbox.WEB);
      sbx2.initialize("template.html");
      sbx2.applyRule(new Rule.Commit(this, "jQuery"));
      sbx2.applyRule(new Rule.Commit(this, "$"));
    </script>
    </head>
  <body>
    <!-- Body of the page -->
    <h1 id="headline">Headline</h1>
    <script type="text/javascript">
      window.$("#headline").text("Changed Headline");
      document.getElementById("headline").innerHTML=sbx2.dom.document.
          getElementById("headline").innerHTML;
    </script>
  </body>
</html>
```

Figure 10 Execution of a web library in a sandbox. The first <*script*> tag loads the sandbox implementation. The body of the second <*script*> tag instantiates a new sandbox and initializes the sandbox with a predefined HTML template (see Figure 11). Later it commits intended effects to the application state and copies data from the sandbox internal DOM.

```
<!DOCTYPE html>
  <html lang="en">
  <head>
    <script src="jquery.js"></script>
    <script src="jquery.formatDateTime.js"></script>
  </head>
  <body>
    <!-- Body of the page -->
    <h1 id="headline">Headline</h1>
  </body>
</html>
```

Figure 11 File *template.html* contains the <*script*> tags for loading the *jQuery* code from *index.html* in Figure 7.

```
<!DOCTYPE html>
  <html lang="en">
  <head>
    <!-- ... -->
    <!-- Checks for conflicts with Datejs -->
    <script type="text/javascript">
      sbx2.applyRule(
        new Rule.Commit(Date, function(sbx, effect) {
          return !sbx.inConflictWith(sbx2, Date);
        });
      sbx2.applyRule(
      new Rule.RollbackOn(Date, function(sbx, effect) {
          return sbx.inConflictWith(sbx2, Date);
        });
    </script>
    </head>
  <body">
    <!-- ... -->
  </body>
</html>
```

Figure 12 Checking for conflicts. The HTML code first checks for conflicts between *Datejs* and *jQuery* before it commits the modification of the library or rolls back.

```
<!DOCTYPE html>
  <html lang="en">
  <head>
    <!-- DecentJS code-->
    <script src="decent.js"></script>
    <!-- Runs Datejs in a fresh sandbox. -->
    <script type="text/javascript">
      var sbx = new Sandbox(this, Sandbox.DEFAULT);
      sbx.request("datejs.js");
      sbx.applyRule(new Rule.CommitOn(Date, function(sbx, effect) {
            return (effect instanceof Effect.Set) && !(effect.name in Date);
      }));
    </script>
    <!-- Runs jQuery in a fresh sandbox. -->
    <script type="text/javascript">
      var sbx2 = new Sandbox(this, Sandbox.WEB);
      sbx2.initialize("template.html");
      sbx2.applyRule(new Rule.Commit(this, "jQuery"));
      sbx2.applyRule(new Rule.Commit(this, "$"));
    </script>
    <!-- Checks for conflicts with Datejs -->
    <script type="text/javascript">
      sbx2.applyRule(new Rule.Commit(Date, function(sbx, effect) {
            return !sbx.inConflictWith(sbx2, Date);
        });
      sbx2.applyRule(new Rule.RollbackOn(Date, function(sbx, effect) {
            return sbx.inConflictWith(sbx2, Date);
        });
    </script>
  </head>
  <body>
    <!-- Body of the page -->
    <h1 id="headline">Headline</h1>
    <script type="text/javascript">
      window.$("#headline").text("Changed Headline");
      document.getElementById("headline").innerHTML=sbx2.dom.document.
            getElementById("headline").innerHTML;
    </script>
  </body>
</html>
```

Figure 13 Execution of library code in a sandbox (cf. Section A in the paper). The first <***script***> tag loads the sandbox implementation. The second <***script***> tag instantiates a new sandbox *sbx* and loads and executes *Datejs* inside the sandbox. Later it commits intended effects to the native *Date* object. The third <***script***> tag instantiates sandbox *sbx2* and initializes the sandbox with a predefined HTML template (see Figure 11 in the paper). Later it commits intended modifications to the application state. The last <***script***> tag checks for conflicts between *Datejs* and *jQuery* before it commits further modification on *Date* or rolls back. The <***script***> tag included in the body performs a modification of the sandbox internal DOM and copies the changes to the global DOM,

B Application Scenarios

This section considers some example scenarios that use the implemented system. All examples are drawn from other projects and use this work's sandboxing mechanism to guarantee noninterference.

B.1 TreatJS

TreatJS [20] is a higher-order contract system for JavaScript which enforces contracts by run-time monitoring. TreatJS is implemented as a library so that all aspects of a contract can be specified using the full JavaScript language.

For example, the base contract *typeNumber* checks its argument to be a number.

```
var typeNumber = Contract.Base(function (arg) {
  return (typeof arg) === 'number';
});
```

Asserting a base contracts to a value causes the predicate to be checked by applying the predicate to the value. In JavaScript, any function can be used as any return value can be converted to boolean[21].

```
Contract.assert(1, typeNumber); // accepted
```

TreatJS relies on the sandbox presented in this work to guarantee that the execution of contract code does not interfere with the contract abiding execution of the host program.

As read-only access to objects and functions is safe and useful in many contracts, TreatJS facilitates making external references visible inside of the sandbox.

For example, the *isArray* contract below references the global object *Array*.

```
var isArray = Contract.With({Array:Array}, Contract.Base(function (arg) {
  return (arg instanceof Array);
}));
```

However, TreatJS forbids all write accesses and traps the unintended write to the global variable *type* in the following code.

```
var typeNumberBroken = Contract.Base(function(arg) {
  type = (typeof arg);
  return type === 'number';
});
```

B.2 TreatJS Online

TreatJS-Online[22] [22] is a web frontend for experimentation with the TreatJS contract system [20]. It enables the user to enter code fragments that run in combination with the TreatJS code. All aspects of TreatJS are accessible to the user code. However, the user code should neither be able to compromise the contract system nor the website's functioning by writing to the browser's *document* or *window* objects. Without any precaution, a code snippet like

[21] JavaScript programmers speak of *truthy* or *falsy* about values that convert to true or false.
[22] `http://www2.informatik.uni-freiburg.de/~keilr/treatjs/`

```
function Observer(target, handler) {
  var sbx = new Sandbox({}, {/* parameters omitted */});
  var controller = {
    get: function(target, name, receiver) {
      var trap = handler.get;
      var result = trap && sbx.call(trap, target, name, receiver);
      var raw = target[name];
      return observerOf(raw, result) ? result : raw;
    }};
    return new Proxy(target, controller);
  }
```

Figure 14 Implementation of an observer proxy (excerpt). The *get* trap evaluates the user specific trap in a sandbox to guarantee noninterference. Afterwards it performs the usual operation and compares the outcomes of both executions. Other traps can be implemented in the same way.

```
Contract.assert = function(arg) {
  return arg;
}
```

could change the *Contract* objects to influence subsequent executions,

To avoid these issues, the website creates a fresh sandbox environment, builds a function closure with the user's input, and executes the user code in the sandbox. The sandbox grants read-only access to the TreatJS API and to JavaScript's built-in objects like *Object*, *Function*, *Array*, and so on, but it does not provide access to browser objects like *document* and *window*. Further, each new invocation reverts the sandbox to its initial state.

B.3 Observer Proxies

An observer proxy[23] is a restricted version of a JavaScript proxy that cannot change the behavior of the proxy's target arbitrarily. It implements a projection in that it either implements the same behavior as the target object or it raises an exception. A similar feature is provided by Racket's chaperones [37].

Such an observer can cause a program to fail more often, but in case it does not fail it would behave in the same way as if not observer were present.

Figure 14 contains the getter part of the JavaScript implementation of *Observer*, the constructor of an observer proxy. It accepts the same arguments as the constructor of a normal proxy object. It returns a proxy, but interposes a different handler, *controller*, that wraps the execution of all user provided traps in a sandbox.

The controller's *get* trap evaluates the user's *get* trap (if one exists) in a sandbox. Next, it performs a normal property access on the target value to produce the same side effects and to obtain a baseline value to compare the results. *observerOf* checks whether the sandboxed result is suitably related to the baseline value.

[23] This observer proxy in this Subsection should not be confused with the *observer proxy* mention in the paper. The observer mentions in Section 3.5 is a normal proxy implementing a membrane.

$$
\begin{array}{lll}
\mathit{Constant} & \ni c & \\
\mathit{Variable} & \ni x, y & \\
\mathit{Expression} & \ni e, f, g ::= & c \mid x \mid op(e, f) \mid \lambda x.e \mid e(f) \\
 & & \mid \mathit{new}\ e \mid e[f] \mid e[f] = g
\end{array}
$$

Figure 15 Syntax of λ_J.

$$
\begin{array}{llll}
\mathit{Value} & \ni u, v, w & ::= & c \mid l \\
 & & & \\
\mathit{Closure} & \ni f & ::= & - \mid (\rho, \lambda x.e) \\
\mathit{Dictionary} & \ni d & ::= & \emptyset \mid d[c \mapsto v] \\
\mathit{Object} & \ni o & ::= & (d, f, v) \\
 & & & \\
\mathit{Environment} & \ni \rho & ::= & \emptyset \mid \rho[x \mapsto v] \\
\mathit{Store} & \ni \sigma & ::= & \emptyset \mid \sigma[l \mapsto o]
\end{array}
$$

Figure 16 Semantic domains of λ_J.

C Semantics of Sandboxing

This section first introduces λ_J, an untyped call-by-value lambda calculus with objects and object proxies that serves as a core calculus for JavaScript, inspired by previous work [13, 21]. It defines its syntax and describes its semantics informally. Later on we extends λ_J to a new calculus λ_J^{SBX}, which adds a sandbox to the core calculus.

C.1 Core Syntax of λ_J

Figure 15 defines the syntax of λ_J. A λ_J expression is either a constant, a variable, an operation on primitive values, a lambds abstraction, an application, a creation of an empty object, a property read, or a property assignment. Variables x, y are drawn from denumerable sets of symbols and constants c include JavaScript's primitive values like numbers, strings, booleans, as well as *undefined* and *null*.

The syntax do not make proxies available to the user, but offers an internal method to wrap objects.

$$
\begin{array}{lll}
\mathit{Sandbox} & \ni \mathcal{S} & ::= \Lambda x.e \\
 & & \\
\mathit{Expression} & \ni e, f, g & ::= \cdots \mid \mathcal{S} \mid \mathit{fresh}\ e \\
\mathit{Term} & \ni t & ::= \cdots \mid \mathit{fresh}\ \mathcal{S} \mid \mathit{wrap}(v) \\
 & & \\
\mathit{Object} & \ni o & ::= \cdots \mid (l, \widehat{l}, \widehat{\rho}) \\
\mathit{Values} & \ni u, v, w & ::= \cdots \mid (\widehat{\rho}, \mathcal{S})
\end{array}
$$

Figure 17 Extensions of λ_J^{SBX}.

$$\begin{array}{lll} \mathit{Term} & \ni t ::= & e \\ & & \mid op(v, f) \mid op(v, w) \mid l(f) \mid l(v) \mid \mathit{new}\ v \\ & & \mid l[f] \mid l[c] \mid l[f] = g \mid l[c] = g \mid l[c] = w \end{array}$$

Figure 18 Intermediate terms of λ_J.

C.2 Semantic Domains

Figure 16 defines the semantic domains of λ_J.

Its main component is a store that maps a location l to an object o, which is a native object (non-proxy object) represented by a triple consisting of a dictionary d, a potential function closure f, and a value v acting as prototype. A dictionary d models the properties of an object. It maps a constant c to a value v. An object may be a function in which case its closure consists of a lambda expression $\lambda x.e$ and an environment ρ that binds the free variables. It maps a variable x to a value v. A non-function object is indicated by $-$ in this place.

A value v is either a constant c or a location l.

C.3 Evaluation of λ_J

A pretty-big-step semantics [4] introduces intermediate terms to model partially evaluated expressions (Figure 18). An intermediate term is thus an expression where zero or more top-level subexpressions are replaced by their outcomes.

The evaluation judgment is similar to a standard big-step evaluation judgment except that its input ranges over intermediate terms: It states that evaluation of term t with initial store σ, and environment ρ results in a final store σ' and value v.

$$\rho \vdash \langle \sigma, t \rangle \Downarrow \langle \sigma', v \rangle$$

Figure 19 defines the standard evaluation rules for expressions e in λ_J. The inference rules for expressions e are mostly standard. Each rule for a composite expression evaluates exactly one subexpression and then recursively invokes the evaluation judgment to continue. Once all subexpressions are evaluated, the respective rule performs the desired operation.

C.4 Sandboxing of λ_J

This section extends the base calculus λ_J to a calculus λ_J^{SBX} which adds sandboxing of function expressions. The calculus describes only the core features that illustrates the principles of our sandbox. Further features of the application level can be implemented in top of the calculus.

Figure 17 defines the syntax and semantics of λ_J^{SBX} as an extension of λ_J. Expressions now contain a sandbox abstraction $\mathcal{S}$ and a sandbox construction *fresh* e that instantiates a fresh sandbox.

Terms are extended with a *fresh* $\mathcal{S}$ term. A new internal *wrap*(v) term, which did not occour in source programs, wraps a value in a sandbox environment.

Objects now contain object proxies. A proxy object is a single location controlled by a proxy handler that mediates the access to the target location. For simplification, we represent handler objects by there meta-data. So, each handler is an sandbox handler that enforces write-protection (viz. by an *secure* location $\widehat{l}$ that acts as an shadow object for the proxies target object l and a single *secure* environment $\widehat{\rho}$).

Const
$$\rho \vdash \sigma, c \Downarrow \sigma, c$$

Var
$$\rho \vdash \sigma, x \Downarrow \sigma, \rho(x)$$

Op-E
$$\frac{\rho \vdash \sigma, e \Downarrow \sigma', v \qquad \rho \vdash \sigma', \mathit{op}(v, f) \Downarrow \sigma'', w}{\rho \vdash \sigma, \mathit{op}(e, f) \Downarrow \sigma'', w}$$

Op-F
$$\frac{\rho \vdash \sigma, f \Downarrow \sigma', u \qquad \rho \vdash \sigma', \mathit{op}(v, u) \Downarrow \sigma'', w}{\rho \vdash \sigma, \mathit{op}(v, f) \Downarrow \sigma'', w}$$

Op
$$\frac{w = \mathit{op}(v, u)}{\rho \vdash \sigma, \mathit{op}(v, u) \Downarrow \sigma, w}$$

Abs
$$\frac{l \notin \mathit{dom}(\sigma) \qquad \sigma' = \sigma[l \mapsto (\emptyset, (\rho, \lambda x.e), \mathit{null})]}{\rho \vdash \sigma, \lambda x.e \Downarrow \sigma', l}$$

App-E
$$\frac{\rho \vdash \sigma, e \Downarrow \sigma', l \qquad \rho \vdash \sigma', l(f) \Downarrow \sigma'', w}{\rho \vdash \sigma, e(f) \Downarrow \sigma'', w}$$

App-F
$$\frac{\rho \vdash \sigma, f \Downarrow \sigma', v \qquad \rho \vdash \sigma', l(v) \Downarrow \sigma'', w}{\rho \vdash \sigma, l(f) \Downarrow \sigma'', w}$$

App
$$\frac{(d, (\rho', \lambda x.e), u) = \sigma(l) \qquad \rho'[x \mapsto v] \vdash \sigma, e \Downarrow \sigma', w}{\rho \vdash \sigma, l(v) \Downarrow \sigma', w}$$

New-E
$$\frac{\rho \vdash \sigma, e \Downarrow \sigma', v \qquad \rho \vdash \sigma', \mathit{new}\ v \Downarrow \sigma'', w}{\rho \vdash \sigma, \mathit{new}\ e \Downarrow \sigma'', w}$$

New
$$\frac{l \notin \mathit{dom}(\sigma) \qquad \sigma' = \sigma[l \mapsto (\emptyset, -, v)]}{\rho \vdash \sigma, \mathit{new}\ v \Downarrow \sigma', l}$$

Get-E
$$\frac{\rho \vdash \sigma, e \Downarrow \sigma', l \qquad \rho \vdash \sigma', l[f] \Downarrow \sigma'', w}{\rho \vdash \sigma, e[f] \Downarrow \sigma'', w}$$

Get-F
$$\frac{\rho \vdash \sigma, f \Downarrow \sigma', c \qquad \rho \vdash \sigma', l[c] \Downarrow \sigma'', w}{\rho \vdash \sigma, l[f] \Downarrow \sigma'', w}$$

Get
$$\frac{(d, f, v) = \sigma(l) \qquad c \in \mathit{dom}(d)}{\rho \vdash \sigma, l[c] \Downarrow \sigma, d(c)}$$

Get-Proto
$$\frac{(d, f, l') = \sigma(l) \qquad c \notin \mathit{dom}(d) \qquad \rho \vdash \sigma, l'[c] \Downarrow \sigma, v}{\rho \vdash \sigma, l[c] \Downarrow \sigma, v}$$

Get-Undef
$$\frac{(d, f, c') = \sigma(l) \qquad c \notin \mathit{dom}(d)}{\rho \vdash \sigma, l[c] \Downarrow \sigma, \mathit{undefined}}$$

Put-E
$$\frac{\rho \vdash \sigma, e \Downarrow \sigma', l \qquad \rho \vdash \sigma', l[f] = g \Downarrow \sigma'', w}{\rho \vdash \sigma, e[f] = g \Downarrow \sigma'', w}$$

Put-F
$$\frac{\rho \vdash \sigma, f \Downarrow \sigma', c \qquad \rho \vdash \sigma', l[c] = g \Downarrow \sigma'', w}{\rho \vdash \sigma, l[f] = g \Downarrow \sigma'', w}$$

Put-G
$$\frac{\rho \vdash \sigma, g \Downarrow \sigma', v \qquad \rho \vdash \sigma', l[c] = v \Downarrow \sigma'', w}{\rho \vdash \sigma, l[c] = g \Downarrow \sigma'', w}$$

Put
$$\frac{(d, f, u) = \sigma(l) \qquad \sigma' = \sigma[l \mapsto (d[c \mapsto v], f, u)]}{\rho \vdash \sigma, l[c] = v \Downarrow \sigma', v}$$

Figure 19 Inference rules for intermediate terms of λ_J.

$$\textsc{Sandbox-Fresh-E}\quad \frac{\rho \;\vdash\; \sigma, e \;\Downarrow\; \sigma', \Lambda x.f \qquad \rho \;\vdash\; \sigma', \mathit{fresh}\ \Lambda x.f \;\Downarrow\; \sigma'', v}{\rho \;\vdash\; \sigma, \mathit{fresh}\ e \;\Downarrow\; \sigma'', v}$$

$$\textsc{Sandbox-Fresh}\quad \frac{\emptyset \;\vdash\; \sigma, \Lambda x.e \;\Downarrow\; \sigma, (\widehat{\rho}, \Lambda x.e)}{\rho \;\vdash\; \sigma, \mathit{fresh}\ \Lambda x.e \;\Downarrow\; \sigma, (\widehat{\rho}, \Lambda x.e)}$$

$$\textsc{Sandbox-Abstraction}\quad \widehat{\rho} \;\vdash\; \sigma, \Lambda x.e \;\Downarrow\; \sigma, (\widehat{\rho}, \Lambda x.e)$$

$$\textsc{Sandbox-Application}\quad \frac{\widehat{\rho} \;\vdash\; \sigma, \mathit{wrap}(v) \;\Downarrow\; \sigma', \widehat{v} \qquad \widehat{\rho}[x \mapsto \widehat{v}] \;\vdash\; \sigma', e \;\Downarrow\; \sigma'', \widehat{w}}{\rho \;\vdash\; \sigma, (\widehat{\rho}, \Lambda x.e)(v) \;\Downarrow\; \sigma'', \widehat{w}}$$

Figure 20 Sandbox abstraction and application rules of λ_J^{SBX}.

For clarity, we write $\widehat{v}$, $\widehat{u}$, $\widehat{w}$ for wrapped values that are imported into a sandbox, $\widehat{\rho}$ for a sandbox environment that only contains wrapped values, and $\widehat{l}$ for locations of proxies and shadow objects.

Consequently, values are extended with sandboxes which represents an sandbox expression wrapped in a sandbox environment that is to be executed when the value is used in an application.

C.4.1 Evaluation of λ_J^{SBX}

Figure 20 contains its inference rules for sandbox abstraction and sandbox application of λ_J^{SBX}. The formalization employs pretty-big-step semantics [4] to model side effects while keeping the number of evaluation rules manageable.

The rule for expression *fresh* e (Rule Sandbox-Fresh-E) evaluates the subexpression and invokes the evaluation judgment to continue. The other rules show the last step in a pretty big step evaluation. Once all subexpressions are evaluated, the respective rule performs the desired operation.

Sandbox execution happens in the context of a secure sandbox environment to preserve noninterference. So a sandbox definition (abstraction) will evaluate to a sandbox closure containing the sandbox expression (the abstraction) together with an empty environment (Rule Sandbox-Fresh). Each sandbox executions starts from a fresh environment. This guarantees that not unwrapped values are reachable by the sandbox.

Sandbox abstraction (Rule Sandbox-Abstraction) proceeds only on secure environments, which is either an empty set or an environment that contains only secure (wrapped) values.

Sandbox execution (Rule Sandbox-Application) applies after the first expression evaluates to a sandbox closure and the second expression evaluates to a value. It wraps the given value and triggers the evaluation of expressions e in the sandbox environment $\widehat{\rho}$ after binding the wrapped value $\widehat{v}$. Value $\widehat{v}$ acts as the global object of the sandbox. It can be used to make values visible inside ob the sandbox.

C.4.2 Sandbox Encapsulation

The sandbox encapsulation (Figure 21) distinguishes several cases. A primitive value and a sandbox closure is not wrapped.

To wrap a location that points to a non-proxy object, the location is packed in a fresh proxy along with a fresh shadow object and the current sandbox environment. This packaging ensures that each further access to the wrapped location has to use the current environment.

$$\textsc{Wrap-Const}\qquad \widehat{\rho} \vdash \sigma, \mathit{wrap}(c) \Downarrow \sigma, c$$

$$\textsc{Wrap-Sandbox}\qquad \widehat{\rho} \vdash \sigma, \mathit{wrap}((\widehat{\rho}', \mathcal{S})) \Downarrow \sigma, (\widehat{\rho}', \mathcal{S})$$

$$\textsc{Wrap-NonProxyObject}\qquad \frac{\begin{array}{c} \nexists l' \in \mathit{dom}(\sigma):\ (l, \widehat{l}, \widehat{\rho}) = \sigma(l') \\ \widehat{\rho} \vdash \sigma, \mathit{compile}(l) \Downarrow \sigma', \widehat{l} \\ \widehat{l}' \notin \mathit{dom}(\sigma') \qquad \sigma'' = \sigma'[\widehat{l}' \mapsto (l, \widehat{l}, \widehat{\rho})] \end{array}}{\widehat{\rho} \vdash \sigma, \mathit{wrap}(l) \Downarrow \sigma'', \widehat{l}'}$$

$$\textsc{Wrap-Existing}\qquad \widehat{\rho} \vdash \sigma[\widehat{l} \mapsto (l, \widehat{l}', \widehat{\rho})], \mathit{wrap}(l) \Downarrow \sigma, \widehat{l}$$

$$\textsc{Wrap-ProxyObject}\qquad \widehat{\rho} \vdash \sigma[\widehat{l} \mapsto (l, \widehat{l}', \widehat{\rho})], \mathit{wrap}(\widehat{l}) \Downarrow \sigma, \widehat{l}$$

Figure 21 Inference rules for sandbox encapsulation.

$$\textsc{Recompile-NonFunctionObject}\qquad \frac{\widehat{l} \notin \mathit{dom}(\sigma) \qquad \sigma' = \sigma[\widehat{l} \mapsto (\emptyset, -, \mathit{null})]}{\widehat{\rho} \vdash \sigma[l \mapsto (d, -, v)], \mathit{compile}(l) \Downarrow \sigma, \widehat{l}}$$

$$\textsc{Recompile-FunctionObject}\qquad \frac{\begin{array}{c} \nexists \widehat{l} \in \mathit{dom}(\sigma):\ (d, (\widehat{\rho}, \lambda x.e), v) = \sigma(\widehat{l}) \\ \widehat{l} \notin \mathit{dom}(\sigma) \qquad \sigma' = \sigma[\widehat{l} \mapsto (\emptyset, (\widehat{\rho}, \lambda x.e), \mathit{null})] \end{array}}{\widehat{\rho} \vdash \sigma[l \mapsto (d, (\rho, \lambda x.e), v)], \mathit{compile}(l) \Downarrow \sigma, \widehat{l}}$$

$$\textsc{Recompile-Existing}\qquad \widehat{\rho} \vdash \sigma[\widehat{l} \mapsto (d, (\widehat{\rho}, \lambda x.e), v)], \mathit{compile}(\widehat{l}) \Downarrow \sigma, \widehat{l}$$

$$\textsc{Recompile-ProxyObject}\qquad \frac{\widehat{\rho} \vdash \sigma[l \mapsto (l', \widehat{l}, \widehat{\rho})], \mathit{compile}(l') \Downarrow \sigma, \widehat{l}}{\widehat{\rho} \vdash \sigma[l \mapsto (l', \widehat{l}, \widehat{\rho})], \mathit{compile}(l) \Downarrow \sigma, \widehat{l}}$$

Figure 22 Inference rules for object re-compilation.

In case the location is already wrapped by a sandbox proxy or the location of a sandbox proxy gets wrapped then the location to the existing proxy is returned. This rule ensures that an object is wrapped at most once and thus preserves object identity inside the sandbox.

The shadow object is build from recompiling (Figure 22) the target object. A shadow objects is a new empty object that may carry a sandboxed replication of its closure part.

For a non-function object, recompiling returns an empty object that later on acts as a sink for property assignments on the wrapped objects.

For a function object, recompiling extracts the function body from the closure and redefines the body with respect to the current sandbox environment. The new closure is put into a new empty object. This step erases all external bindings of function closure and ensures that the application of a wrapped function happens in the context of the secure sandbox environment.

In case the function is already recompiled, function recompilation returns the existing replication.

C.4.3 Application, Read, and Assignment

Function application, property read, and property assignment distinguish two cases: either the operation applies directly to a non-proxy object or it applies to a proxy. If the target of the operation is not a proxy object, then the usual rules apply.

Figure 23 contains the inference rules for function application and property access for the

$$\textsc{App-Sandbox}\quad \frac{\begin{array}{c}\widehat{\rho} \vdash \sigma, \mathit{wrap}(v) \Downarrow \sigma', \widehat{v} \\ \rho \vdash \sigma', \widehat{l}(\widehat{v}) \Downarrow \sigma'', \widehat{w}\end{array}}{\rho \vdash \sigma[l \mapsto (l', \widehat{l}, \widehat{\rho})], l(v) \Downarrow \sigma'', \widehat{w}}$$

$$\textsc{Get-Shadow}\quad \frac{c \in \mathit{dom}(\widehat{l}) \qquad \rho \vdash \sigma[l \mapsto (l', \widehat{l}, \widehat{\rho})], \widehat{l}[c] \Downarrow \sigma', \widehat{v}}{\rho \vdash \sigma[l \mapsto (l', \widehat{l}, \widehat{\rho})], l[c] \Downarrow \sigma', \widehat{v}}$$

$$\textsc{Get-Sandbox}\quad \frac{\begin{array}{c}c \notin \mathit{dom}(\widehat{l}) \qquad \rho \vdash \sigma[l \mapsto (l', \widehat{l}, \widehat{\rho})], l'[c] \Downarrow \sigma', v \\ \widehat{\rho} \vdash \sigma', \mathit{wrap}(v) \Downarrow \sigma'', \widehat{v}\end{array}}{\rho \vdash \sigma[l \mapsto (l', \widehat{l}, \widehat{\rho})], l[c] \Downarrow \sigma'', \widehat{v}}$$

$$\textsc{Put-Sandbox}\quad \frac{\rho \vdash \sigma[l \mapsto (l', \widehat{l}, \widehat{\rho})], \widehat{l}[c] = v \Downarrow \sigma', v}{\rho \vdash \sigma[l \mapsto (l', \widehat{l}, \widehat{\rho})], l[c] = v \Downarrow \sigma', v}$$

Figure 23 Inference rules for function application, property read, and property assignment.

non-standard cases.

The application of a wrapped function proceeds by unwrapping the function and evaluating it in the sandbox environment contained in the proxy. The function argument and its result are known to be wrapped in this case.

A property read on a wrapped object has two cases depending on if the accessed property has been written in the sandbox before, or not. The notation $c \in \mathit{dom(l)}$ is defined as an shortcut of a dictionary lookup $c \in \mathit{dom(d)}$ with $\sigma(l) = (d, f, v)$.

A property read of an affected field reads the property from the shadow location. Otherwise, it continues the operation on the target and wraps the resulting value. An assignment to a wrapped object is continues with the operation on the shadow location $\widehat{l}$.

In JavaScript, write operations do only change properties of the object's dictionary. They do not affect the object's prototype. Therefor, the shadow object did not contain any prototype informations. It acts only a shadow that absorbs write operations.

D Technical Results

As JavaScript is a memory safe programming language, a reference can be seen as the right ti modify the underlying object. If an expressions body can be shown not to contain unprotected references to objects, then it cannot modify this data.

To prove soundness of our sandbox we show termination insensitive noninterference. It requires to show that the initial store σ of a sandbox application is *observational equivalent* to the final store σ', that remains after the application. In detail, the sandbox application may introduce new objects or even write to shadow objects (only reachable inside of the sandbox) but every value reachable from the outside remains unmodified.

As the calculus in Appendix C did not support variable updates on environments ρ the only way to make changes persistent is to modify objects. Thus, proving noninterference relates different stores and looks for differences in the store with respect to all reachable values.

D.1 Observational Equivalence on Stores

First, we introduce an equivalence relation on stores with respect to other semantic elements.

▶ **Definition 1.** Two stores σ, σ' are equivalent w.r.t constants c, c' if the constants are equal.

$$(\sigma, c) \simeq (\sigma', c') \Leftrightarrow c = c$$

▶ **Definition 2.** Two stores σ, σ' are equivalent w.r.t locations l, l' if they are equivalent on the location's target.

$$(\sigma, l) \simeq (\sigma', l') \Leftrightarrow (\sigma, \sigma(l)) \simeq (\sigma', \sigma'(l'))$$

▶ **Definition 3.** Two stores σ, σ' are equivalent w.r.t non-proxy objects (d, f, v), (d', f', v') if they are equivalent on the objects's constituents.

$$(\sigma, (d, f, v)) \simeq (\sigma', (d', f', v')) \Leftrightarrow$$

$$(\sigma, d) \simeq (\sigma', d') \wedge (\sigma, f) \simeq (\sigma', f') \wedge (\sigma, v) \simeq (\sigma', v')$$

▶ **Definition 4.** Two stores σ, σ' are equivalent w.r.t dictionaries d, d' if they are equivalent on the dictionary's content.

$$(\sigma, d) \simeq (\sigma', d') \Leftrightarrow$$

$$\mathit{dom(d)} = \mathit{dom(d')} \wedge \forall c \in \mathit{dom(d)}.(\sigma, d(c)) \simeq (\sigma', d'(c))$$

▶ **Definition 5.** Two stores σ, σ' are equivalent w.r.t closures $(\rho, \lambda x.e)$, $(\rho', \lambda x.f)$ if the are equivalent on the closure's environment and both abstractions are equal.

$$(\sigma, (\rho, \lambda x.e)) \simeq (\sigma', (\rho', \lambda x.f)) \Leftrightarrow$$

$$(\sigma, \rho) \simeq (\sigma', \rho') \wedge \lambda x.e = \lambda x.f$$

▶ **Definition 6.** Two stores σ, σ' are equivalent w.r.t environments ρ, ρ' if the are equivalent on the environment's content.

$$(\sigma, \rho) \simeq (\sigma', \rho') \Leftrightarrow$$

$$\mathit{dom(\rho)} = \mathit{dom(\rho')} \wedge \forall x \in \mathit{dom(\rho)}.(\sigma, \rho(x)) \simeq (\sigma', \rho(x))$$

▶ **Definition 7.** Two stores σ, σ' are equivalent w.r.t proxy objects $(l, \widehat{l}, \widehat{\rho})$, $(l', \widehat{l}', \widehat{\rho}')$ if they are equivalent on the objects's constituents.

$$(\sigma, (l, \widehat{l}, \widehat{\rho})) \simeq (\sigma', (l', \widehat{l}', \widehat{\rho}')) \Leftrightarrow$$

$$(\sigma, l) \simeq (\sigma', l') \wedge (\sigma, \widehat{l}) \simeq (\sigma', \widehat{l}') \wedge (\sigma, \widehat{\rho}) \simeq (\sigma', \widehat{\rho}')$$

▶ **Definition 8.** Two stores σ, σ' are equivalent w.r.t sandbox closures $(\widehat{\rho}, \Lambda x.e)$, $(\widehat{\rho}', \Lambda x.f)$ if the are equivalent on the sandbox's environment and both abstractions are equal.

$$(\sigma, (\widehat{\rho}, \Lambda x.e)) \simeq (\sigma', (\widehat{\rho}', \Lambda x.f)) \Leftrightarrow$$

$$(\sigma, \widehat{\rho}) \simeq (\sigma', \widehat{\rho}') \wedge \Lambda x.e = \Lambda x.f$$

Now, the observational equivalence for stores can be states as follows.

▶ **Definition 9.** Two stores σ, σ' are observational equivalent under environment ρ if they are equivalent on all values $v \in \{\rho(x) \mid x \in \mathit{dom}(\rho)\}$

$$\sigma \simeq_\rho \sigma' \Leftrightarrow \forall x \in \mathit{dom}(\rho).(\sigma, \rho(x)) \simeq (\sigma', \rho(x))$$

▶ **Lemma 10.** *Suppose that* $\rho_i \vdash \langle \sigma_i, e \rangle \Downarrow \langle \sigma'_i, v_i \rangle$ *then for all* σ_j, ρ_j *with* $(\sigma_i, \rho_i) \simeq (\sigma_j, \rho_j)$. $\rho_j \vdash \langle \sigma_j, e \rangle \Downarrow \langle \sigma'_j, v_j \rangle$ *with* $(\sigma'_i, \rho_i) \simeq (\sigma'_j, \rho_j)$ *and* $(\sigma'_i, v) \simeq (\sigma'_j, w)$.

Proof. Proof by induction on the derivation of e. ◀

D.2 Noninterference

▶ **Theorem 11.** *For each* ρ *and* σ *with* $\rho \vdash \langle \sigma, (\mathit{fresh}\ \Lambda x.e)(f) \rangle \Downarrow \langle \sigma', v \rangle$ *it holds that* $\sigma \simeq_\rho \sigma'$.

Proof. Proof by induction on the derivation of e. ◀

E Related work

There is a plethora of literature on securing JavaScript, so we focus on the distinguishing features of our sandbox and on related work not already discussed in the body of the paper.

Sandboxing JavaScript

The most closely related work to our sandbox mechanism is the design of access control wrappers for revocable references and membranes [39, 26]. In a memory-safe language, a function can only cause effects to objects reachable from references in parameters and global variables. A revocable reference can be instructed to detach from the objects, so that they are no longer reachable and safe from effects. However, as membranes by themselves do not handle side effects (every property access can be the call of a side-effecting getter) they do not implement a sandbox in the way we did.

Agten et al. [2] implement a JavaScript sandbox using proxies and membranes. As in our work, they place wrappers around sensitive data (e.g., DOM elements) to enforce policies and to prevent the application state from unprotected script inclusion. However, instead of encapsulating untrusted code they require that scripts are compliant with SES [34], a subset of JavaScript's "strict mode" that prohibits features that are either unsafe or that grant uncontrolled access, and use an SES-library to execute those scripts. A language-embedded JavaScript parser transforms non-compliant scripts at run time, but doing so restricts the handling of dynamic code compared to our approach.

TreatJS, a JavaScript contract system [20], uses a sandboxing mechanism similar to the sandbox presented in this work to guarantee that the execution of a predicate does not interfere with the execution of a contract abiding host program. As in our work, they use JavaScript's dynamic facilities to traverse the scope chain when evaluating predicates and they use JavaScript proxies to make external references visible when evaluating predicates.

Arnaud et al. [3] provide features similar to the sandboxing mechanism of TreatJS [20]. Both approaches focus on access restriction to guarantee side-effect free contract assertion. However, neither of them implements a full-blown sandbox, because writing is completely forbidden and always leads to an exception.

Our sandbox works in a similar way and guarantees read-only access to target objects, but redirects write operations to shadow objects such that local modifications are only visible inside the sandbox. However, access restrictions in all tree approaches affect only values that cross the border between two execution environments. Values that are defined and used inside, e.g. local values, were not restricted. Write access to those values is fine.

Patil et al. [29] present *JCShadow*, a reference monitor implemented as a Firefox extension. Their tool provides fine-grained access control to JavaScript resources. Similar to DecentJS, they implement shadow scopes that isolate scripts from each other and which regulate the granularity of object access. Unlike DecentJS, *JCShadow* achieves a better runtime performance. While more efficient, their approach is platform-dependent as it is tied to a specific engine and requires active maintenance to keep up with the development of the enigine. DecentJS, in contrast, is a JavaScript library based on the reflection API, which is part of the standard.

Most other approaches (e.g., [11, 27, 8, 1]) implement restrictions by filtering and rewriting untrusted code or by removing features that are either unsafe of that grant uncontrolled access. For exampe, Caja [11, 27] compiles JavaScript code in a sanitized JavaScript subset that can safely be executed on normal engines. Because static guarantees do not apply to code created at run time using ***eval*** or other mechanisms, Caja restricts dynamic features and rewrites the code to a "cajoled" version with additional run-time checks that prevent access to unsafe function and objects.

Static approaches come with a number of drawbacks, as shown by a number of papers [23, 9, 31]. First, they either restrict the dynamic features of JavaScript or their guarantees simply do not apply to code generated at run time. Second, maintenance requires a lot of effort because the implementation becomes obsolete as the language evolves.

Thus, dynamic effect monitoring and dynamic access restriction plays an important role in the context of JavaScript security, as shown by a number of authors [3, 39, 26, 19].

Effect Monitoring

Richards et al. [32] provide a WebKit implementation to monitor JavaScript programs at run time. Rather than performing syntactic checks, they look at effects for history-based access control and revoke effects that violate policies implemented in C++.

Transcript, a Firefox extension by Dhawan et al. [7], extends JavaScript with support for transactions and speculative DOM updates. Similar to DecentJS, it builds a transactional scope and permits the execution of unrestricted guest code. Effects within a transaction are logged for inspection by the host program. They also provide features to commit updates and to recover from effects of malicious guest code.

JSConTest [16] is a framework that helps to investigate the effects of unfamiliar JavaScript code by monitoring the execution and by summarizing the observed access traces to access permission contracts. It comes with an algorithm [17] that infers a concise effect description from a set of access paths and it enables the programmer to specify the effects of a function using access permission contracts.

JSConTest is implemented by an offline source code transformation. Because of JavaScrip's flexibility it requires a lot of effort to construct an offline transformation that guarantees full interposition and that covers the full JavaScript language. This, the implementation of JSConTest has known omissions: no support for ***with*** and prototypes, and it does not apply to code created at run time using ***eval*** or other mechanisms.

JSConTest2 is a redesign and a reimplementation of JSConTest using JavaScript proxies. The new implementation addresses shortcomings of the previous version: it guarantees full interposition for the full language and for all code regardless of its origin, including dynamically loaded code and code injected via ***eval***.

JSConTest2 [18] monitors read and write operations on objects through access permission contracts that specify allowed effects. A contract restricts effects by defining a set of permitted access paths starting from some anchor object. However, the approach works differently. JSConTest2 has to encapsulate sensitive data instead of encapsulating dubious functions.

Language-embedded Systems

JSFlow [15] is a full language-embedded JavaScript interpreter that enforces information flow policies at run time. Like DecentJS, *JSFlow* itself is implemented in JavaScript. Compared to DecentJS, the *JSFlow* interpreter causes a significantly higher run-time impact than the our sandbox, which only reimplements the JavaScript semantics on the membrane.

A similar slowdown is reported for js.js [38], another language-embedded JavaScript interpreter conceived to execute untrusted JavaScript code. Its implementation provides a wealth of powerful features similar to DecentJS: fine-grained access control, support for the full JavaScript language, and full browser compatibility. However, its average slowdown in the range of 100 to 200 is significantly higher than DecentJS's.

F Evaluation Results

This section reports on our experience with applying the sandbox to select programs. We focus on the influence of sandboxing on the execution time.

We use the Google Octane Benchmark Suite[24] to measure the performance of the sandbox implementation. Octane measures a JavaScript engine's performance by running a selection of complex and demanding programs (benchmark programs run between 5 and 8200 times).

Google claims that Octane "measure[s] the performance of JavaScript code found in large, real-world web applications, running on modern mobile and desktop browsers. Each benchmark is complex and demanding .

We use Octane as it is intended to measure the engine's performance (benchmark programs run between 5 and 8200 times). we claim that it is the heaviest kind of benchmark. Every real-world library (e.g. *jQuery*) is less demanding and runs without an measurable runtime impact.

Octane 2.0 consists of 17 programs[25] that range from performance tests to real-world web applications (Figure **??**), from an OS kernel simulation to a portable PDF viewer. Each program focuses on a special purpose, for example, function and method calls, arithmetic and bit operations, array manipulation, JavaScript parsing and compilation, etc.

F.1 Testing Procedure

All benchmarks were run on a machine with two AMD Opteron processors with 2.20 GHz and 64 GB memory. All example runs and measurements reported in this paper were obtained with the SpiderMonkey JavaScript engine.

For benchmarking, we wrote a new start script that loads and executes each benchmark program in a fresh sandbox. By setting the sandbox global to the standard global object, we ensure that each benchmark program can refer to properties of the global object as needed.

As sandboxing wraps the global object in a membrane it mediates the interaction of the benchmark program with the global application state.

All run time measurements were taken from a deterministic run, which requires a predefined number of iterations[26], and by using a warm-up run.

F.2 Results

Figure 24 and Figure 25 contains the run-time statistics for all benchmark programs in two different configurations, which are explained in the figure's caption, and lists the readouts of some internal counters. Multiple read effects to the same field of an object are counted as one effect.

As expected, the run time increases when executing a benchmark in a sandbox. While some programs like *EarleyBoyer*, *NavierStrokes*, *pdf.js*, *Mandreel*, and *Box2DWeb* are heavily affected, others are only slightly affected: *Richards*, *Crypto*, *RegExp*, and *Code loading*, for instance. Unfortunately, *DeltaBlue* and *zlib* do not run in our sandbox. *DeltaBlue* attempts to add a new property to the global *Object.prototype* object. As our sandbox prevents unintended modifications to *Object.prototype* the new property is only visible inside of the

[24] `https://developers.google.com/octane`
[25] `https://developers.google.com/octane/benchmark`
[26] Programs run either for one second or for a predefined number of iterations. If there are too few iterations in one second, it runs for another second.

Benchmark	**Baseline**	**Sandbox w/o Effects**		**Sandbox w Effects**	
	time (sec)	*time (sec)*	*slowdown*	*time (sec)*	*slowdown*
Richards	9 sec	12 sec	1.33	15 sec	1.67
DeltaBlue	9 sec	-	-	-	-
Crypto	18 sec	42 sec	2.33	88 sec	4.89
RayTrace	9 sec	74 sec	8.22	498 sec	55.33
EarleyBoyer	19 sec	202 sec	10.63	249 sec	13.11
RegExp	6 sec	9 sec	1.5	12 sec	2
Splay	3 sec	19 sec	6.33	33 sec	11
SplayLatency	3 sec	19 sec	6.33	33 sec	11
NavierStokes	3 sec	56 sec	18.67	61 sec	20.33
pdf.js	7 sec	113 sec	16.14	778 sec	111.14
Mandreel	8 sec	151 sec	18.88	483 sec	60.38
MandreelLatency	8 sec	151 sec	18.88	483 sec	60.38
Gameboy Emulator	4 sec	17 sec	4.25	26 sec	6.50
Code loading	8 sec	11 sec	1.38	12 sec	1.50
Box2DWeb	4 sec	145 sec	36.25	1,302 sec	325.50
zlib	7 sec	-	-	-	-
TypeScript	26 sec	61 sec	2.35	328 sec	12.62
Total	135 sec	1.082 sec	8.01	4,401 sec	32.60

Figure 24 Timings from running the Google Octane 2.0 Benchmark Suite. The first column **Baseline** gives the baseline execution times without sandboxing. The column **Sandbox w/o Effects** shows the time required to complete a sandbox run without effect logging and the relative slowdown (Sandbox time/Baseline time). The column **Sandbox w Effects** shows the time and slowdown (w.r.t. Baseline) of a run with fine-grained effect logging.

current sandbox and only to objects created with ***new*** *Object()* and *Object.create()*, but not to those created using object literals.

The *zlib* benchmark uses an indirect call[27] to ***eval*** to write objects to the global scope, which is not allowed by the ECMAScript 6 (ECMA-262) specification. Another benchmark, *Code loading*, also uses an indirect call to ***eval***. A small modification makes the program compatible with the normal ***eval***, which can safely be used in our sandbox.

In the first experiment we turn off effect logging, whereas in the second one it remains enabled. Doing so separates the performance impact of the sandbox system (proxies and shadow objects) from the impact caused by the effect system. From the running times we find that the sandbox itself causes an average slowdown of 8.01 (over all benchmarks).

Our experimental setup wraps the global object in a membrane and mediates the interaction between the benchmark program and the global application state. As each benchmark program contains every source required to run the benchmark in separation, except global objects and global functions, the only thing that influences the execution time is read/write access to global elements.

In absolute times, raw sandboxing causes a run time deterioration of 0.003ms per sandbox operation (effects) (0.011ms with effect logging enabled). For example, the *Box2DWeb* benchmark requires 145 seconds to complete and performs 132,722,198 effects on its membrane. Its baseline requires 4 seconds. Thus, sandboxing takes an additional 141 seconds. Hence, there is an overhead of 0.001ms per operation (0.010ms with effect logging enabled).

[27] An indirect call invokes the ***eval*** function by using a name other than ***eval***.

Benchmark	**Objects**	**Effects**	**Size of Effect List**		
			Reads	*Writes*	*Calls*
Richards	14	492073	20	2	5
DeltaBlue	-	-	-	-	-
Crypto	21	4964248	29	2	11
RayTrace	18	51043282	26	3	8
EarleyBoyer	33	4740377	42	8	6
RegExp	16	296995	23	2	6
Splay	16	1635732	23	2	8
SplayLatency	16	1635732	23	2	8
NavierStokes	15	4089	21	2	6
pdf.js	36	77665629	59	8	21
Mandreel	31	39948598	50	2	21
MandreelLatency	31	39948598	50	2	21
Gameboy Emulator	28	1225935	42	2	16
Code loading	12417	107481	50	2	13
Box2DWeb	28	132722198	38	2	14
zlib	-	-	-	-	-
TypeScript	23	27518481	34	2	9
Total	12743	383949448	530	43	173

Figure 25 Numbers from internal counters. Column **Objects** shows the numbers of wrap objects and column **Effects** gives the total numbers of effects. Column **Size of Effect List** lists the numbers of *different* effects after running the benchmark. Column **Reads** shows the number of read effects distinguished from there number of write effects (Column **Writes**) and distinguished from there number of call effects (Column **Calls**). Multiple effects to the same field of an object are counted as one effect.

The results from the tests also indicate that the garbage collector runs more frequently, but there is no significant increase in the memory consumption. For the effect-heaviest benchmark *Box2D* we find that the *virtual memory size* increases from 157MByte (raw run) to 197MB (full run with effect logging).

However, when looking at all benchmarks, the difference in the virtual memory size compared with the baseline run ranges from -126MByte to +40MByte for a raw sandbox run without effect logging and from -311MByte to +158MByte for a full run with fine grained effect logging. Appendix F.4 shows the memory usage of the different benchmark programs and their difference compared with the baseline.

F.3 Google Octane Scores Values

Octane reports its result in terms of a score. The Octane FAQ[28] explains the score as follows: "*In a nutshell: bigger is better. Octane measures the time a test takes to complete and then assigns a score that is inversely proportional to the run time.*" The constants in this computation are chosen so that the current overall score (i.e., the geometric mean of the individual scores) matches the overall score from earlier releases of Octane and new benchmarks are integrated by choosing the constants so that the geometric mean remains the same. The rationale is to maintain comparability.

[28] `https://developers.google.com/octane/faq`

Benchmark	Isolation	Effects	Baseline
Richards	4825	4135	6552
DeltaBlue	-	-	6982
Crypto	2418	1131	5669
RayTrace	1387	179	9692
EarleyBoyer	954	767	10345
RegExp	911	1139	1535
Splay	1268	713	8676
SplayLatency	3630	1818	12788
NavierStokes	989	890	15713
pdf.js	434	63	8182
Mandreel	346	106	9102
MandreelLatency	2518	526	12526
Gameboy Emulator	6572	4780	31865
Code loading	7348	6000	9136
Box2DWeb	453	50.1	18799
zlib	-	-	42543
TypeScript	4554	792	12588

Figure 26 Scores for the Google Octane 2.0 Benchmark Suite (bigger is better). Block **Isolation** contains the score values of a raw sandbox run without effect logging, whereas block **Effects** contains the score values of a full run with fine-grained effect logging. The last column **Baseline** gives the baseline scores without sandboxing.

Figure 26 contains the scores of all benchmark programs in different configurations, which are explained in the figure's caption. All scores were taken from a deterministic run, which requires a predefined number of iterations[29], and by using a warm-up run.

As expected, all scores drop when executing the benchmark in a sandbox. In the first experiment, we turn off effect logging, whereas the second run is with effect logging. This splits the performance impact into the impact caused by the sandbox system (proxies and shadow objects) and the impact caused by effect system.

F.4 Memory Consumption

Figure 27, Figure 28, and Figure 29 shows the memory consumption recorded when running the Google Octane 2.0 Benchmark Suite. The numbers indicate that there is no significant increase in the memory consumed. For example, the difference of the virtual memory size ranges from -126 to 40 for a raw sandbox run and from -311 to +158 for a full run with fine grained effect logging.

[29] Programs run either for one second or for a predefined number of iterations. If there are to few iterations in one second, it runs for another second.

Benchmark	**Baseline**			
	Virtual	**Resident**	**Text/Data**	**Shared**
	size	*size*	*size*	*size*
Richards	134	19	109	5
DeltaBlue	-	-	-	-
Crypto	225	105	201	6
RayTrace	148	31	124	5
EarleyBoyer	500	363	476	6
RegExp	226	108	202	6
Splay	535	416	511	6
SplayLatency	535	416	511	6
NavierStokes	141	24	116	5
pdf.js	316	169	292	6
Mandreel	305	182	280	6
MandreelLatency	305	182	280	6
Gameboy Emulator	194	62	170	6
Code loading	268	142	243	6
Box2DWeb	157	53	132	5
zlib	-	-	-	-
TypeScript	473	369	448	6

Figure 27 Memory usage when running the Google Octane 2.0 Benchmark Suite without sandboxing. Column **Virtual** shows the virtual memory size, column **Resident** shows the resident set size, column **Text/Data** shows the Text/Data segment size, and column **Text/Data** shows the Text/Data segment size. All values are in MByte.

Benchmark	**Sandbox w/o Effects**							
	Virtual		**Resident**		**Text/Data**		**Shared**	
	size	*diff.*	*size*	*diff.*	*size*	*diff.*	*size*	*diff.*
Richards	135	1	20	1	110	1	5	0
DeltaBlue	-	-	-	-	-	-	-	-
Crypto	232	+7	106	+1	208	+7	6	0
RayTrace	148	+0	31	+0	124	+0	6	+1
EarleyBoyer	374	-126	272	-91	350	-126	6	0
RegExp	224	-2	107	-1	200	-2	6	0
Splay	466	-69	352	-64	422	-89	6	0
SplayLatency	466	-69	352	-64	422	-89	6	0
NavierStokes	134	-7	18	-6	109	-7	5	0
pdf.js	274	-42	123	-46	250	-42	6	0
Mandreel	263	-42	133	-49	239	-41	5	-1
MandreelLatency	263	-42	133	-49	239	-41	5	-1
Gameboy Emulator	188	-6	60	-2	163	-7	6	0
Code loading	259	-9	138	-4	234	-9	6	0
Box2DWeb	197	+40	97	+44	172	+40	6	+1
zlib	-	-	-	-	-	-	-	-
TypeScript	424	-49	325	-44	428	-20	6	0

Figure 28 Memory usage of a raw sandbox run without effect logging. Column **Virtual** shows the virtual memory size, column **Resident** shows the resident set size, column **Text/Data** shows the Text/Data segment size, and column **Text/Data** shows the Text/Data segment size. Sub-column *size* shows the size in MByte and sub-column *diff.* shows the difference to the baseline (Sandbox size - Baseline size) in MByte.

Benchmark	Sandbox w Effects							
	Virtual		**Resident**		**Text/Data**		**Shared**	
	size	*diff.*	*size*	*diff.*	*size*	*diff.*	*size*	*diff.*
Richards	167	33	54	35	142	33	6	1
DeltaBlue	-	-	-	-	-	-	-	-
Crypto	209	-16	93	-12	184	-17	6	0
RayTrace	181	+33	63	+32	157	+33	6	+1
EarleyBoyer	189	-311	86	-277	195	-281	6	0
RegExp	224	-2	104	-4	200	-2	6	0
Splay	424	-111	321	-95	399	-112	6	0
SplayLatency	424	-111	321	-95	399	-112	6	0
NavierStokes	134	-7	19	-5	109	-7	5	0
pdf.js	272	-44	116	-53	248	-44	6	0
Mandreel	347	+42	160	-22	323	+43	6	0
MandreelLatency	347	+42	160	-22	323	+43	6	0
Gameboy Emulator	214	+20	84	+22	190	+20	6	0
Code loading	262	-6	139	-3	238	-5	6	0
Box2DWeb	191	+34	72	+19	166	+34	6	+1
zlib	-	-	-	-	-	-	-	-
TypeScript	631	+158	493	+124	607	+159	6	0

Figure 29 Memory usage of a full run with fine-grained effect logging. Column **Virtual** shows the virtual memory size, column **Resident** shows the resident set size, column **Text/Data** shows the Text/Data segment size, and column **Text/Data** shows the Text/Data segment size. Sub-column *size* shows the size in MByte and sub-column *diff.* shows the difference to the baseline (Sandbox size - Baseline size) in MByte.