Transparent Object Proxies for JavaScript

Matthias Keil[1], Sankha Narayan Guria[2], Andreas Schlegel[1], Manuel Geffken[1], and Peter Thiemann[1]

1 **Institute for Computer Science**
University of Freiburg
Freiburg, Germany
{keilr,schlegea,geffken,thiemann}@informatik.uni-freiburg.de

2 **Indian Institute of Technology Jodhpur**
Jodhpur, India
sankha@iitj.ac.in

Abstract

Proxies are the swiss army knives of object adaptation. They introduce a level of indirection to intercept select operations on a target object and divert them as method calls to a handler. Proxies have many uses like implementing access control, enforcing contracts, virtualizing resources.

One important question in the design of a proxy API is whether a proxy object should inherit the identity of its target. Apparently proxies should have their own identity for security-related applications whereas other applications, in particular contract systems, require transparent proxies that compare equal to their target objects.

We examine the issue with transparency in various use cases for proxies, discuss different approaches to obtain transparency, and propose two designs that require modest modifications in the JavaScript engine and cannot be bypassed by the programmer.

We implement our designs in the SpiderMonkey JavaScript interpreter and bytecode compiler. Our evaluation shows that these modifications of have no statistically significant impact on the benchmark performance of the JavaScript engine. Furthermore, we demonstrate that contract systems based on wrappers require transparent proxies to avoid interference with program execution in realistic settings.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases JavaScript, Proxies, Equality, Contracts

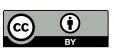
Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.149

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.1.1.2

1 Introduction

A proxy modifies the functionality of an underlying target object by introducing a level of indirection that intercepts all operations on the target. As all problems in computer science can be solved by another level of indirection[1], proxies may be called the Swiss army knives of object adaptation. Indeed, proxies are widely used to perform resource management, to access remote objects, to impose access control [20, 13], to implement contract checking [19, 10, 5], to restrict the functionality of an object [19], to enhance the interface of an object [22], to

[1] A famous quote by David Wheeler.



29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 149–173

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
var target = { /* some object */ };
var handler = { /* empty handler */ };
var proxy = new Proxy (target, handler);
proxy===target; // evaluates to false
```

Listing 1 Comparing a proxy with its target. The methods of *handler* determine the behavior of *proxy*. If *handler* is empty, then *proxy* behaves exactly like *target*.

implement dynamic effect systems, as well as for meta-level extension, behavioral reflection, security, and concurrency control [16, 2, 4].

A proxy implementation provides an intercession API that enables the programmer to trap all operations on the target object (with few exceptions). Further, a program should not be able to distinguish a proxy from a non-proxy object so that putting a proxy in place of an object does not affect the outcome of the program (save for the new behavior introduced by the proxy). For that reason, the JavaScript Proxy API [20], a part of the current ECMAScript 6 draft standard, does not include a function that checks whether an object is a proxy, it does not provide traps for all operations on objects, and it restricts some traps to avoid breaking certain object invariants [21].

However, the JavaScript Proxy API embodies a design decision that reveals the presence of proxies in some important use cases. This decision concerns object equality. The API description[2] says: *The double and triple equal (==, ===) operator is not trapped. p1 === p2 if and only if p1 and p2 refer to the same proxy.* The standard does not even mention proxies in the definition of object equality: *If x and y are the same Object value, return true.* [9, Section 7.2.13] In other words, proxies are *opaque*, which means that each proxy has its own identity, different from all other (proxy or non-proxy) objects. Given opaque proxies, an equality test can be used to distinguish a proxy from its target as demonstrated in Listing 1.

Even though *target* and *proxy* behave identically, they are not considered equal. Thus, in a program that uses object equality, the introduction of a proxy along one execution path may change the meaning of the program without even invoking an operation on the proxy (which may behave differently from the same operation on the target).

Equality for opaque proxies is straightforward to implement and works well under the assumption that proxies and their targets are never part of the same execution environment. For example, the revocable membrane pattern [20] enables to safely pass object references to untrusted code, control their operation on these objects, and revoke all access rights afterwards. This pattern is implemented using proxies and it partitions the execution environment into security realms so that the objects that live in the same realm are never in a proxy-target relationship. By this convention, the situation outlined in Listing 1 never arises inside a compartment.

But the assumption that proxies never share their execution environment with their targets is not always appropriate. One prominent use case is the implementation of a contract system. A contract system provides a domain specific language to state very precise type-like assertions for values in an untyped language. Two examples for such systems are the contract framework of Racket [11, Chapter 7] and Contracts.js for JavaScript [5]. Both systems implement contracts on objects with specific wrapper objects, Racket's chaperones or impersonators [19] and JavaScript proxies, respectively.

[2] `https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Proxy`

```
1  // contract wrapper implementation
2  function checkPredicate (pred) {
3    return {
4      set: function(target, prop, val) {
5        if (!pred (val)) { throw new ContractException(); };
6        target[prop] = val;
7      }
8    }
9  };
10 function assertContract(target, pred) {
11   return new Proxy (target, checkPredicate(pred));
12 }
13 // application code
14 function addBonus (acc1, acc2, amount) {
15   acc1.balance += amount;
16   if (acc1 !== acc2) { // test objects for equality
17     acc2.balance += amount;
18   }
19 }
20 var account = { balance: 10 };
21 var restricted = assertContract (account, function(x) {
22   return (x >= 0);
23 });
24 addBonus (account, account, 40); // raises account by 40
25 addBonus (restricted, account, 40); // raises account by 80
```

Listing 2 Application with contract wrapper.

Listing 2 contains a JavaScript implementation of a simple contract wrapper. The implementation uses JavaScript proxies, which are introduced in more detail in Section 2. The wrapper applies to a JavaScript object and it enforces that all property values written to the object fulfill a predicate *pred*. Otherwise, the wrapper raises an exception.

The call *assertContract (target, pred)* returns a proxy for the target object with a handler created by the function call *checkPredicate (pred)*. Whenever a property *prop* is set on the proxy, the method *set* of its handler is invoked with the target object, the property *prop*, and the new value as arguments. The handler throws an exception if the predicate *pred* is not fulfilled, otherwise it performs the set operation on the target.

The application code contains an *addBonus* function that takes two accounts and a bonus amount to add. The intention is to give a bonus to each account once. Thus, if the two accounts are different, then the balance of the second account must be adjusted, too.

The last couple of lines create an account and a restricted handle to the same account, where the restricted handle does not permit the account to overdraw. In line 24, a bonus of *40* is added to *account* and *account*. This bonus addition executes correctly because the equality test in line 16 yields *false*. However, performing the bonus addition with the restricted version and the standard account leads to adding a bonus of *80* to *account* because the test in line 16 yields *true*.

This example shows that the introduction of a contract monitor like *assertContract* may change the semantics of a program even in cases where the contract is not exercised. But this

change in behavior violates a ground rule for monitoring: a monitor should never interfere with a program conforming to the monitored property. (In Section 3, we make a similar case for access restricting membranes.)

While this particular example is constructed we demonstrate in Section 7 that such situations do occur in practice. Racket programmers have also run into this issue[3], as chaperones and impersonators behave opaquely with respect to Racket's `eq?` operation.

As a remedy, we propose alternative designs for *transparent proxies* that are better suited for use cases such as certain contract wrappers and access restricting membranes. We evaluate these designs with respect to usability. We further implement them in the Firefox SpiderMonkey JavaScript engine and evaluate the impact of transparent proxies on benchmark performance.

Overview and contributions

Section 2 introduces the JavaScript Proxy API, explains the membrane pattern, and sketches the implementation of a contract system based on proxies. Section 3 discusses different use cases of proxies and assesses them with respect to the requirements on proxy transparency. Section 4 contains an in-depth discussion of the programmer's expectation from an equality operation and how it would be affected by the design of a proxy API. Section 5 presents alternative designs to obtain transparency. **We present two novel designs for transparent proxies that do not impede the implementation of advanced proxy management.** In Section 6, we describe how **we implement these two designs in the Firefox JavaScript engine (interpreter and baseline JIT compiler)**. In Section 7, **we demonstrate that our modification to the JavaScript engine does not affect benchmark performance**. Furthermore, **we present evidence that the danger of interference for a contract implementation based on opaque proxies is real**: it arises in instrumented benchmark programs, not just in artificially constructed examples. Section 8 discusses related work and Section 9 concludes.

The technical report [12] accompanying this paper contains an appendix with a detailed descriptions of the algorithms used in our implementation. The implementation of the JavaScript engine with transparent proxies is available in a Github repository[4].

2 Proxies, Membranes, and Contracts

This section introduces the JavaScript Proxy API and presents two typical applications of proxies: membranes that regulate access to an object network and contracts that check assertions on the values manipulated by a program.

2.1 Proxies

A proxy is an object intended to be used in place of a *target object*. As the target object may also be a proxy, we call the unique non-proxy object that is transitively reachable through a chain of proxies the *base target* for each proxy in this chain. The behavior of a proxy is controlled by a *handler object*, which may modify the original behavior of the target object in many respects. A typical use case is to have the handler mediate access to the target object, but in JavaScript the handler has a full range of intercession facilities that we only touch on.

[3] Personal communication with Robby Findler, February 2015.
[4] `https://github.com/sankha93/js-tproxy`

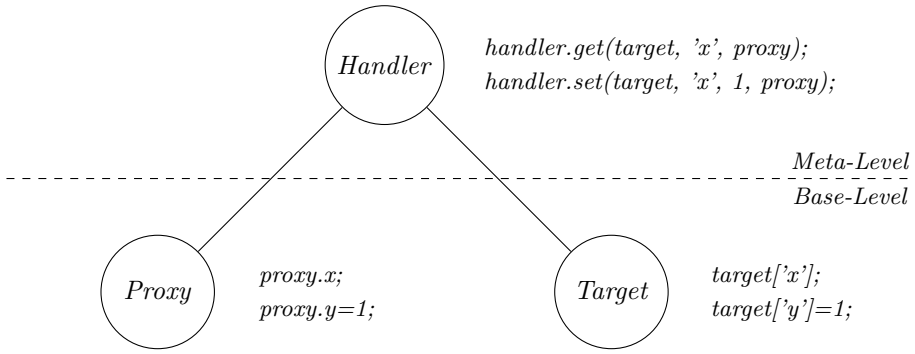


Figure 1 Example of a proxy operation.

The JavaScript Proxy API [20] contains a proxy constructor that takes the designated target object and a handler object:

```
var target = { /* some properties */};
var handler = { /* trap functions... */ };
var proxy = new Proxy (target, handler);
```

The handler object contains optional trap functions that are called when the corresponding operation is performed on the proxy. Operations like property read, property assignment, and function application are forwarded to the 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 functionality if the trap is not specified.

Performing an operation like property get or property set on the proxy object results in a meta-level call to the corresponding trap on the handler object. For example, the property get operation *proxy.x* invokes *handler.get(target,'x',proxy)* and the property set operation *proxy.y=1* invokes the trap *handler.set(target,'y',1,proxy)*, if these traps are present.

Figure 1 illustrates this situation with a handler that forwards all operations to the target.

However, a handler may redefine or extend the semantics of an operation arbitrarily. For example, a handler may forward a property access to its target object only if the property is not locally present. The following example demonstrates a handler that implements a copy-on-write policy for its target by intercepting all write operations and serving reads on them locally. Thus, reading *target.a* at the end may return a different value than *42*.

```
function makeHandler() {
  var local = {};
  return {
    get: function(target, name, receiver) {
      return (name in local) ? local[name] : target[name];
    },
    set: function(target, name, value, receiver) {
      return local[name]=value;
    }
  };
}
var child = new Proxy (target, makeHandler());
child.a = 42; // does not change target
```

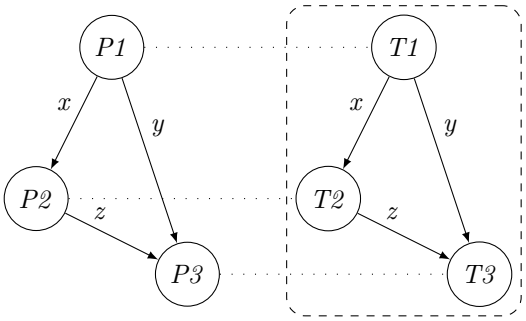


Figure 2 Property access through membrane.

Proxy and handler objects are based on the JavaScript Proxy API [20, 21], which is part of the JavaScript draft standard ES6. This API is implemented in Firefox since version 18.0 and in Chrome V8 since version 3.5.

JavaScript's proxies are *opaque*: each proxy object has its own identity different from all other (proxy) objects. The proxy identity is observable with the JavaScript equality operators == and ===: When applied to two objects, both operators compare object identities.[5]

The following example (which continues the preceding code fragment) illustrates this behavior. Comparing distinct proxies returns false even though the underlying target is the same. Similarly, the target object is different from any of its proxies.

```
42 var proxy2 = new Proxy (target, handler);
43 (proxy==proxy2); // false
44 (proxy==target); // false
```

We already mentioned in the introduction that equality cannot be trapped. While there are good reasons for this design decision [21], we mention in passing that it would be hard to design and implement an efficient equality trap because equality is a binary method.

2.2 Membranes

A *membrane* is a regulated communication channel between an object and the rest of the program. It ensures that all objects reachable from an object behind the membrane are also behind the membrane. Figure 2 shows a membrane (dashed line) around targets *T1*, *T2*, and *T3* implemented by the wrapper objects *P1*, *P2*, and *P3*. Each property access through the wrapper (e.g., *P1.x*) returns a wrapper for *T1.x*, which is created on demand. After installing the membrane, no *new* direct references to target objects behind the membrane become available. This mechanism may be used to revoke all references to an object network at once or to enforce write protection on the objects behind the membrane [20, 17].

An *identity preserving membrane* is a membrane that furthermore guarantees that no target object has more than one proxy. Thus, proxy identity outside the membrane reflects target object identity inside. That is, if *T1.x.z===T1.y*, then also *P1.x.z===P1.y*. Figure 2 depicts such an identity preserving membrane.

Both kinds of membrane may be implemented with the JavaScript Proxy API and a weak map that associates target objects with their proxies [20].

[5] If one argument has a primitive type, == attempts to convert the other argument to the same primitive type, whereas === returns false if the types are different. If both arguments are objects, then both operators do the same.

2.3 Contracts

Dynamically checked software contracts lie at the core of Meyer's *Design by Contract*™ methodology [15]. A contract specifies the interface of a software component by stating obligations and benefits for the component's implementors and users. Contracts state invariants for objects as well as preconditions and postconditions for functions and methods. Contracts are particularly important for dynamically typed languages as these languages do not provide static guarantees beyond memory safety. For such languages, contract systems are indispensable tools to create maintainable software.

A contract may specify a property of a value. In many cases, a simple assertion suffices, but many interesting properties of functions and objects cannot be checked immediately. For example, the contract *Num→Num* expresses that a function maps a number to a number. It can only be checked when the function is called: the caller must provide a number argument and then the result must be a number, too. Similarly, a check that a certain object property must always be assigned a number can only be checked when actually setting the property.

We call such contracts on functions and objects *delayed contracts*, because their assertion never signals a contract violation immediately. The standard implementation of a delayed contract is by wrapping the function/object in a proxy. The proxy's handler implements traps to mediate the use of the function/object and to assert its contract when the function is called or the object is read or written to.

The following example sketches the implementation of a contract assertion for *Num→Num*. It installs a proxy where the handler supplies an *apply* trap that gets invoked when the proxy is called as a function. The arguments to *apply* are the target object, the ***this*** object, and an array containing the arguments.

```
45 var handler = {
46   apply: function(target, thisArg, argsArray) {
47     if (typeof argsArray[0] !== 'number') { throw new Exception (); }
48     var result = target.apply(thisArg, argsArray);
49     if (typeof result !== 'number') { throw new Exception (); }
50     return result;
51   }
52 };
53 var addOne = function (x) { return x+1; };
54 var addOneNN = new Proxy (addOne, handler);
```

Thus, the monitored function *addOneNN* is implemented by a proxy with a suitable handler. The contract systems Contract.js [5] and TreatJS [14] are both implemented in this way. As JavaScript does not support "proxification" (i.e., the transformation of an arbitrary object into a proxy), it is conceivable that *addOne*, the original object, and *addOneNN*, the proxy, are both accessible in the same execution environment.

3 Opacity vs. Transparency for Proxies

In this section, we question whether JavaScript proxies need be opaque by considering various use cases for proxies and evaluating whether they could be served equally well with transparent proxies, where equality is defined as equality of base targets.

3.1 Use Case: Object Extension

A common use case of proxies is to extend or redefine the semantics of particular operations on objects. For example, a handler may throw an exception instead of returning *undefined*, it may redirect different operations to different targets (for example to store changes locally or to implement placeholders), it may log or trace operations, or it may notify observers.

In this case, using the proxy may lead to a completely different outcome than using the target object. Thus, proxy and target object should not be confused.

3.2 Use Case: Access Control

Revocable references are the motivating use case for membranes [20, 17]. Instead of passing a target object to an untrusted piece of code, the idea is to pass its proxy wrapped in a protecting membrane. Once the host application deems that the untrusted code has finished its job, it revokes the reference which detaches the proxy from its target. The membrane extends this detachment recursively to all objects reachable from the original target.

Opaque proxies are suitable for implementing membranes as well as their identity preserving variant. However, transparent proxies would work just as well, because the host application only sees original objects whereas the untrusted code only sees proxies. Furthermore, the implementation of revocable references and membranes ensures that there is at most one proxy for each original object. If an execution environment is compartmentalized like this, then each compartment has a consistent view with unique object (or proxy) references, regardless whether proxies are opaque or transparent. In fact, with transparent proxies, a membrane is always identity preserving, the weak map only improves the space efficiency.

3.3 Use Case: Contracts

Proxies implement contracts in Racket [19] and in JavaScript [5, 13, 14]. During maintenance, a programmer may add contracts to a program as understanding improves. To systematically investigate a program in this way, the addition of a new contract must not change a program execution that respects the contract already. In this scenario, the program executes in a mix of original objects and proxy objects. Furthermore, there may be more than one proxy (implementing different contracts) for the same target. If introducing proxies affected the object identity, then some equality comparisons on objects (`eq?` in Racket and ==, *!=*, ===, or *!==* in JavaScript) would flip their outcome, thus changing the semantics.

Our experimental evaluation (Section 7.2) considers a typical program understanding and maintenance scenario where a programmer inserts assertions/contracts to document and validate his/her understanding of the program. We find that mixed (proxy vs. non-proxy) object comparisons occur in realistic programs.

Similar incidents where observed when maintaining Racket's preferences framework. Registering a callback with the framework wrapped the callback in a contract before storing it in an `eq?`-based weak hash table. Because there were no further references to the wrapper, the weak table released it on the next garbage collection. Thus, the callback disappeared mysteriously, leading to unwanted behavior[6]. This problem is evidence that such mixes occur in real situations and also in a system where contracts are aligned with module boundaries.

Thus, we see strong evidence that unintended mixing is hard to avoid even in a well-designed system. If a module has a higher-order interface, then a function passed as a

[6] Personal communication with Robby Findler, February 2015.

parameter may capture an un-proxied version of an object that is also passed as a regular parameter. For example, let T be some delayed contract and consider the module interface

```
55 // foo : (T → boolean, T) → boolean
```

so that *foo* carries a wrapper that asserts this contract. The function accepts two parameters, the first of which is a function. An external caller may use *foo* as follows:

```
56 var x = { /* some object */ };
57 var f = function (y) { return x===y; };
58 foo(f, x);
```

The call to *foo* wraps f and x in the respective contract wrappers for $T \to boolean$ and T. Unfortunately, wrapping the function f does not affect the free variable x in its environment.

Now consider the following contract-abiding implementation of *foo*:

```
59 function foo (f, y) { return f (y); }
```

Inside of *foo*, y is wrapped in the T contract and applying f (wrapped in $T \to boolean$) may wrap it one more time in a T. Thus, in the body of f (line 57), x is unwrapped and y is the same object wrapped at least once in T. Thus, assuming opaque proxies, $x===y$ yields *false*, which is different from the result before installing the contract on *foo* (line 55).

Thus, the existing implementations of higher-order contracts for JavaScript are prone to interfere with the semantics. The situation is similar for Racket. Racket's chaperones and impersonators are opaque because they may be distinguished from their target and from one another using `eq?`. This choice is legitimized by pointing out that the preferred equality test in Racket is the `equal?` function that compares two values for *structural equality*, a non-trivial functionality that is not available out-of-the-box in JavaScript (cf. [1]). Clearly, chaperones and impersonators are transparent with respect to `equal?`. The paper on chaperones and impersonators [19] further acknowledges that "wrappings do affect the identity of objects, as compared with JavaScript's === or Racket's `eq?` comparisons", but remarks that Racket programmers rely much less on object comparison than JavaScript programmers. Indeed, the paper's formalization includes `equal?`, but not `eq?`.

3.4 Assessment

Neither the opaque nor a transparent proxy implementation can be labeled as right or wrong without further qualification. Each is appropriate for particular applications and may lead to undesirable behavior in other applications.

Opaqueness is required for a proxy that changes the behavior of its target significantly. This case corresponds to the use case for impersonators [19].

Transparent proxies can safely be used to implement revocable membranes as well as for other applications that guarantee compartmentalized execution (where proxies and targets never meet). Their use simplifies the implementation of the identity preserving membrane because the weak map from targets to their proxies may be elided. However, the price for this elision is increased space usage for multiple wrappers for the same target. It must be weighed against the time taken to administer a weak target-to-proxy mapping.

Transparency is required for proxies that implement contract wrappers to avoid the interference with the normal program execution pointed out in the introduction and in Section 3.3. To avoid such interference with opaque proxies, a programmer would have to guarantee that contracts are only ever applied to unique object references or that references to the target object do not escape. Such a guarantee may be established by only applying

contracts to objects as they are created or by static analysis. However, the former is an unrealistic assumption and the latter severely limits the freedom that developers want to obtain by using contracts in a dynamically typed language: rather than submitting to a type system, they must submit to a uniqueness or an escape analysis.

A similar case can be made for further wrapper-based programming patterns. For example, a wrapper that records all operations performed on an object reference can be very helpful while debugging. Clearly, such a wrapper must be indistinguishable from its target object.

It is also clear that the behavior of equality is not something that should be left to the whim of the programmer. For example, equality on objects should be an equivalence relation, which means that the equality operations == and === must not be trapped [21].

Thus, the current state of affairs in JavaScript is fully justified, but it is not well suited to implement contract systems. To obtain some insight into a proxy design suitable for implementing contracts, we first explore the important issue of equality and then consider some designs and assess the suitability with respect to the use cases outlined in this section.

4 Invariants for Equality

What does a programmer expect from an equality function on objects? Let's consult the language references for Racket, Java, and JavaScript for cues.

Racket inherits its hierarchy of equality operators from Scheme: `equal?`, `eqv?`, and `eq?`. The Scheme report [18] specifies them as follows: "An equivalence predicate is the computational analogue of a mathematical equivalence relation; it is symmetric, reflexive, and transitive. Of the equivalence predicates described in this section, `eq?` is the finest or most discriminating, `equal?` is the coarsest, and `eqv?` is slightly less discriminating than `eq?`." The intuition is that `equal?` implements structural equality, `eq?` implements pointer equality, and `eqv?` is a compromise between the expensive structural equality[7] and pointer equality, which may distinguish different boxings of the same number.

There are further differences between `eq?` and `equal?`. The function `eq?` is guaranteed to run in $O(1)$, whereas there is no known implementation of `equal?` that runs in less than linear time. Furthermore, `equal?` is not stable in the sense that `(equal? x y)` may hold at some point during execution, but this equality may be destroyed by subsequent assignments in the program. In contracts, `(eq? x y)` does not change as the program proceeds.

Java has an `equals` method in `java.lang.Object`. The documentation of this class reads:[8] "The equals method implements an equivalence relation on non-null object references: ..." It also asks that repeated invocations with the same arguments behave consistently in that they always return the same answer. And finally: "The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`)." About the `==` operator, the JLS says:[9] "The equality operators are commutative ..." and then "the result of `==` is `true` if the operand values are both `null` or both refer to the same object or array; otherwise, the result is `false`." Regarding stability and execution time, the `==` operator is stable and runs in $O(1)$. The `equals` methods is recommended to be implemented in a stable way, but this restriction is not enforced. No bounds on the execution time are prescribed and none can be given.

---

[7] It is supposed to work on cyclic structures, too. [1]
[8] `http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html`
[9] `http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html` in 15.21

The ECMAScript specification [8, Section 11.9] does not mention algebraic properties of the equality operation, but rather contains pseudocode for the abstract equality comparison algorithm, which underlies the == operator. This algorithm implements an operation, which is symmetric, but neither reflexive nor transitive: It is not reflexive because *NaN==NaN* is *false* and it is not transitive because ***new*** *String("foo")=="foo"* and *"foo"==**new** String ("foo")*, but ***new*** *String("foo")==**new** String("foo")* is *false* due to unfortunate interaction with type conversions. Restricted to object arguments the algorithm says: "Return true if x and y refer to the same object. Otherwise, return false." The strict equality comparison algorithm (in Section 11.9.6), which specifies the === operator, is not reflexive because of *NaN*, but appears to be symmetric and transitive. Thus, it is at least a partial equivalence relation. Restricted to objects, both equalities are equivalences if that is implied by sameness.

JavaScript's == operator is not stable: Consider a comparison between an object and a string and then change the *toString* method of the object. However, === is stable because it does not involve type conversion. Restricted to object arguments, both equalities are stable.

The least common denominator for an object equality appears to be that **equality must be a stable equivalence relation**. Often, object equality is explained by alluding to the "sameness" of objects, which is left to further interpretation. In particular, the JLS explicitly mentions that `==` is overly discerning for `String` objects and none of the language specifications addresses proxy objects.

Some proponents of opaque proxies ask that equality should be an observational equivalence. That is, for any conditional of the form

```
60 if (a === b) Statement
```

(where *a* and *b* are variables) it should be possible to substitute *b* for *a* (but respecting lexical binding rules) in *Statement* without changing the semantics. However, in the presence of JavaScript's *with (head){ ... body ... }* statement inside *Statement* further qualifications are needed. As *with* extends the lexical environment by placing *head* on top of the scope chain while executing *body*, a property of *head* may shadow any variables in scope including *a*. Thus, the substitution must not extend to the body of a *with* statement.

If the above conditional appears in the *body* of a *with (head){ ... body ... }* statement, then essentially all bets are off. The object *head* may define *a* or *b* via a getter function or via a proxy handler, which may be impure and return different results on each access. Alternatively, *a* and *b* may be changed by simple assignment to *head*.

Furthermore, the *Statement* may contain a call to *eval*. As this call is executed in the lexical environment of its call site, its execution may perform an assignment to *a* or *b*. As the call to *eval* may be performed indirectly by storing *eval* in a different variable or object property, essentially every function or method call may assign to *a* or *b*.

While ES5 strict mode abolishes the *with* statement and severely restricts the use of *eval*, a call to *eval* may still assign to variables in scope at the point of the call. For example, the following function *f42* always returns *42* because the *eval* assigns to an existing local variable.

```
1 function f42(x) {
2   "use strict";
3   eval("x = 42");
4   return x;
5 }
```

Thus, arriving at a practically useful definition of observational equivalence in JavaScript is quite intricate; perhaps too intricate to be a useful reasoning tool for programmers.

Moreover, we are not aware of a language definition that requires its equality operator to be an observational equivalence.

What is the take-home message from this discussion with respect to the question whether a proxy implementation should be opaque or transparent with respect to equality? We believe that most programming patterns using equality only expect a stable equivalence relation, which is easy to implement for transparent proxies, as we show in Section 5.3. However, disregarding the problems with the *with* statement and *eval*, a weak version of observational equivalence is certainly desirable: For any conditional of the form

```
61 if (a === b) Statement
```

if we substitute b for a in *Statement*, then either the semantics of *Statement* does not change or *Statement* raises an exception.

However, a transparent proxy without further restrictions would not even fulfill weak observational equivalence. As a compromise, the behavioral change of a transparent proxy could be restricted to implement a *projection*, a point that we come back to in Section 5.4.

5 Design Alternatives for Proxy Equality

In this section, we explore some alternatives for designing proxies and equalities and discuss their suitability for the use cases outlined in Section 3.

5.1 Program Rewriting

One way to obtain transparent proxies with an implementation of opaque proxies is to replace all occurrences of ==, !=, ===, and !== with proxy-aware equality functions. These functions can be implemented in JavaScript using a weak map from proxies to their target. The proxy constructor would be extended to maintain this map. It would be possible to treat some proxies as transparent and the rest as opaque.

This approach preserves the existing behavior and retains the possibility to distinguish proxies from target objects in library code implementing proxy abstractions. A macro system like SweetJS [6] may be used to implement such a transformation elegantly, alas, SweetJS is currently implemented as an offline transformation and would need to be extended to deal with `eval` and dynamic loading.

5.2 Additional Equality Operators

Another approach would be to reinterpret the JavaScript equality operators == and === as proxy-transparent and introduce new variants, *:==:* and *:===:*, say, for their opaque cousins (i.e., the current implementations of == and ===). The former operators are used in application code whereas the implementation of proxy abstractions would make use of the opaque operators where needed.

No code transformation is required with this approach. However, it is not clear how to ensure that application code does not use the opaque operators. It is not even clear if it *should not* use them. While proxy abstractions can be implemented, the distinction between application and library seems too brittle.

Given both operations, application code can test if two objects are in a proxy relation with the same target:

```
62 ((x === y) != (x :===: y)); // true, iff x is in a proxy relation to y
```

```
function GetIdentityObject(obj) {
  while(isProxy(obj) && isTransparentProxy(obj)) {
    obj = getProxyTargetObject(obj);
  }
  return obj;
}
```

Listing 3 Pseudo code for GetIdentityObject.

Furthermore, the implementation would either have to use macros, which gives rise to the problems discussed in Section 5.1, or it would have to be implemented in the JavaScript engine, where it requires changes starting in the parser. This point makes it unlikely that such a proposal would be adopted by the community. Moreover, a proliferation of equality operations is confusing for developers as there are already three different kinds of equality in JavaScript: besides equality and strict equality, there is a third equality that fixes reflexivity when comparing *NaN* values.

5.3 Transparent Proxies in the VM

We already discussed that trapping the equality operation is not appropriate. As an alternative, we implement transparent proxies as an extension of a JavaScript VM (cf. Section 6) and provide different constructors for transparent and for opaque proxies. This extension provides a different kind of proxy object on which equality comparison behaves differently. Before testing reference identity as the last step in a comparison of two objects, the equality comparison calls a new internal function *GetIdentityObject* (see Listing 3) that computes the base target of an object. For a non-proxy object, the function returns its argument. For a proxy object, *GetIdentityObject* checks whether the proxy is transparent. If that check fails, then *GetIdentityObject* returns the reference to the current proxy object. Otherwise, it iteratively performs the same checks on the proxy's target. For consistency, the *GetIdentityObject* method also needs to be called in other computations that depend on object identity. One example is the WeakMap abstraction of the ES6 draft standard.

This design enables both scenarios described in Sections 3.2 and 3.3. It also guarantees that equality (on objects) is an equivalence relation.

Transparent proxies need special attention because there are abstractions that require to test whether a reference is a (transparent) proxy. For example, the implementation of access permissions contracts [13] extracts the current permission from a proxy to construct a new proxy with an updated permission. This introspection improves the efficiency of the implementation; its absence would lead to long, inefficient chains of proxy objects.

Thus, for implementing proxy abstractions it is useful to be able to break the transparency. We propose to use secret tokens for this purpose. A token (just an object in JavaScript) stands for a transferable right to perform a particular operation. We attach the token object to a proxy by making it an extra argument to the constructor of transparent proxies, *TransparentProxy*, say. Being a standard object, the token can be hidden in the scope of the function that wraps objects.

```
var wrap = (function() {
  var token = {};
  return function(target, handler) {
    return new TransparentProxy(target, handler, token);
```

```
  };
  // further operations on wrappers
})();
```

Later on, the token can be used to make the transparent proxy visible for equality operations. To this end, we need an equality operation *Object.equals* that takes the token as a third (optional) parameter. The following snippet demonstrates this operation in action.

```
var token = {};
var target = { ... };
var proxyA = new TransparentProxy (target, handlerA, token);
var proxyB = new TransparentProxy (target, handlerB, token);
target == proxyA; // true
proxyA == proxyB; // true
Object.equals(target, proxyA, token); // false: token reveals proxy identity
Object.equals(proxyA, proxyB, token); // false
Object.equals(target, proxyA, { /* some other object */ }); // true
Object.equals(proxyA, proxyB, { /* some other object */ }); // true
```

Weak maps and other internal data structures that depend on object equality may be extended with tokens in the same way.

From different point of view, the tokens assign transparent proxies to distinct realms. Thus, instead of passing tokens one could use object capabilities to create proxies in a particular realm and to create an equality function that only reveals proxies for that realms[10]. A realm constructor may be implemented in JavaScript on top of our token-based implementation.

```
TransparentProxy.createProxyConstructor = function() {
  var token = {};
  var equals = function (x, y) {
    return Object.equals(x, y, token);
  }
  var Constructor = function(target, handler) {
    return new TransparentProxy(target, handler, token);
  }
  return {Constructor:Constructor, equals:equals};
}
```

The realm constructor returns a new transparency realm represented by an object that consists of a fresh constructor for transparent proxies (named *Constructor*) and an *equals* function revealing proxies of that realm.

```
var realm = TransparentProxy.createProxyConstructor();
var proxy1 == realm.Constructor(target, handler);
var proxy2 == realm.Constructor(target, handler);
```

The proxies *proxy1* and *proxy2* are transparent with respect to equality unless someone uses the *realm.equals* method.

```
proxy1 === proxy2; // true
Object.equals(proxy1, proxy2); // true
realm.equals(proxy1, proxy2); // false
```

[10] We would like to thank the anonymous ECOOP 2015 reviewer who suggested this elegant solution.

Here, === and *Object.equals* return *true* and do not reveal proxies. However, the *realm .equals* function is a capability that represents the right to reveal proxies of that realm. Realm-aware weak maps and other internal data structures can be created in same way.

5.4 Observer Proxies

To implement contracts, transparent proxies need not be fully general. It would be sufficient if transparent proxies would be restricted to *Observer*s that implement a projection: they would either return a value identical to the value that would be returned from the target object (this should also include the same side effects) or that restricts the behavior of the target object. An *Observer* can cause a program to fail more often, but otherwise it would behave in the same way as if no observer were present.

A similar feature is provided by Racket's chaperones [19]. A chaperone is a proxy that either returns an identical value, returns a chaperone of this value, or throws an exception. This restricted kind of proxy is shown to be sufficient to implement a contract system. (Recall that chaperones are not transparent.)

The following code snippet sketches the implementation of an *Observer* proxy in JavaScript that mimics Racket's chaperone proxy. The example considers the *get* trap only, but other traps can be implemented in the same way.

```
96  function Observer(target, handler) {
97    var sbx = new Sandbox(/* some parameters */);
98    var controller = {
99      /* further traps omitted */
100     get: function(target, name, receiver) {
101       var result = (trap=handler['get']) ? sbx.call(trap, target, name, receiver) :
              undefined;
102       var raw = target[name];
103       return (result===raw) ? result : raw;
104     }
105   };
106   return new TransparentProxy(target, controller);
107 }
108 var target = { /* some object */ };
109 var handler = {
110   get:function(target, name, receiver) {
111     return target[name];
112   }
113 };
114 var observed = Observer(target, handler);
```

The constructor starts with instantiating a sandbox (line 97). The sandbox is drawn from another contract system for JavaScript, TreatJS [14], that uses membranes and decompilation to implement access restrictions. Functions execute inside the sandbox without interfering with the normal execution.

The implementation distinguishes between a user specified handler (*handler*) whose traps are evaluated in the sandbox to guarantee noninterference and the proxy handler (*controller*) which is used to implement the behavior of the observer. The controller's *get* trap first checks the presence of a *get* trap in the user handler, before it evaluates this trap in the sandbox. Next, it performs a normal property access on the target value. This step is required to

produce the same side effects and to get a reference value to compare the results. Finally, the reference value is compared with the result from calling the trap. Line 103 makes only sense when using transparent proxies. The user specific trap can return an observer of the reference value.

Indeed, the implementation is only correct if one can ensure that *result* is either the *raw* value or a transparent proxy generated by an *Observer*. A user specific handler can simply elude the observers behavior by returning a transparent proxy of the same target but with a different handler object. Correctness can be guaranteed by either hiding transparent proxies from the user level or by using the sandbox to restrict resources access be the handler's trap.

5.5 Recommendation

There is likely no single semantics for object identity that fits the programmers expectation in all possible contexts. A proxy that changes the behavior of its target object significantly needs its own identity and thus needs to be implemented opaquely.

A contract proxy that only restricts the behavior of the original object, we propose to use a transparent observer proxy with the design explained in Section 5.3. For these proxies, === (and friends) will be forwarded to the target objects, recursively. An observer proxy (see Section 5.4) limits the possible change of behavior analogously to chaperones. Technically, observer proxy weakly simulate the original objects.

6 Implementation

We implemented two prototype extensions of the SpiderMonkey JavaScript engine, one according to the design of Section 5.3 and another which extends the proxy handler by an *isTransparent* trap that regulates the proxy's transparency. The first prototype implements a new global object *TransparentProxy* that implements the constructor for transparent proxy objects.

Proxies created with ***new*** *Proxy* in the second prototype are generally opaque, unless they implement an *isTransparent* trap and this trap returns false. Proxies created with ***new*** *TransparentProxy* are generally transparent unless they are compared with *Object.equal* using the correct token. These choices guarantee full backwards compatibility.

The implementation is rather tedious because many things have to be implemented three times as SpiderMonkey consists of an interpreter, the baseline JIT compiler (JaegerMonkey), and the IonMonkey compiler. Support for transparent proxies has to be added at each level because SpiderMonkey switches at run time from interpreter to baseline compiler and then to IonMonkey after a sufficient number of loop iterations. Specifically, the documentation says: "All JavaScript functions start out executing in the interpreter [... which] collects type information for use by the JITs. When a function gets somewhat hot, it gets compiled with JaegerMonkey. [...] When it gets really hot, it is recompiled with IonMonkey." If type information changes, execution falls back all the way to the interpreter.

6.1 JavaScript Interpreter

To cater for transparent proxies, the interpreter had to be changed in a few places.

1. The internal classes `Proxy` and `BaseProxyHandler` had to be extended to support the new *isTransparent* trap.
2. The comparison operators had to be modified to recognize transparent proxies and obtain their base target for comparison,

3. All internal data structures which are connected to the identity of objects (in particular, the map, weak map, and set abstractions of the upcoming JavaScript standard) had to be modified.

6.1.1 JavaScript's Equality Comparison

JavaScript provides two types of comparison operators. The *strict equality comparison* (e.g. ===) returns *false* if the operands have different types. The *equality comparison* (e.g. ==) applies type conversion if the operands have different types; then it essentially performs a strict comparison on the converted values. When comparing two objects x and y, both comparisons behave identically [8, Section 11.9.3]:

> 1.f. "Return *true* if x and y refer to the same object."

Thus, this test for sameness of two objects is only one place where the algorithms for equality comparison and strict equality comparison have to be changed. Our implementation replaces the test (case 1.f. in equality comparison [11.9.3], case 7. in strict equality comparison [11.9.6]) as follows.

1. Let *lhs* be the result of calling *GetIdentityObject* on x.
2. Let *rhs* be the result of calling *GetIdentityObject* on y.
3. Return *true* if *lhs* and *rhs* refer to the same object. Otherwise, return *false*.

6.1.2 Getting the Identity Object

When comparing two objects or when adding an object to a map, transparent proxies do not use their own identity. To get the right identity for the object the operation first checks the transparency of the proxy object and transitively obtains its target object until either an opaque proxy or a native object is reached (cf. Listing 3 for the pseudocode). All object comparisons refer to this internal method.

The actual implementation is slightly more involved, in particular the implementation that supports the *isTransparent* trap (its existence needs to be checked, it needs to be called, and its results needs to be interpreted). Technical details may be checked in the source code which is available in a github repository.

6.1.3 Maps, Sets, and other Data Structures

After modifying the comparison operators the internal data structures *Map*, *Set*, and *WeakMap*, which depend on object equality, have to be adjusted to handle transparent proxies. If *target == proxy* evaluates to *true* then *map.has(target)==map.has(proxy)* should also evaluate to *true*.

When adding a new key-value pair to any Map, WeakMap, or Set, the operation first determines if the key is an object of type *Proxy*. If it is a *Proxy*, then the *GetIdentityObject* internal method is used to determine the identity object; hashing takes place with respect to this identity object, but the original key is stored in the collection along with its value. A subsequent lookup of the identity object or any proxy with the same identity object returns the same stored value. The implementation of the ***for...in*** loop or calling *.keys()* or *.entries()* on a map returns the originally added object as key value.

The example below demonstrates the behavior just described on an empty map object *map*. We first create one transparent and one opaque proxy for the same target. The operation *map.set(target, A)* creates a new map entry, whereas the second one *map.set(proxy1, B);* updates this entry. The third operation *map.set(proxy2, C);* creates a new entry, again.

```
115 var target = {};
116 var proxy1 = new TransparentProxy(target, {});
117 var proxy2 = new Proxy(target, {});
118 map.set(target, A); // map = [target ↦ A]
119 map.set(proxy1, B); // map = [target ↦ B]
120 map.set(proxy2, C); // map = [target ↦ B, proxy2 ↦ C]
```

6.2 Object.equals

Transparent proxies created with *TransparentProxy* are generally indistinguishable from their base target object and from another transparent proxy object of the same target. However, *Object.equals* can be employed to make them distinguishable for algorithms that implement advanced proxy manipulation. To this end, the constructor stores its token argument in a slot of each proxy.

When *Object.equals* is called with arguments *obj1*, *obj2* and an optional argument *secret* the following steps are taken:

- If *secret* is not present, then return the value of *obj1 === obj2*.
- If one of *obj1* or *obj2* is a transparent proxy where the token slot matches the *secret*, then return *true* if *obj1* is the same object as *obj2* (and *false*, otherwise).
- Otherwise return the value of the transparent comparison *obj1 === obj2*.

6.3 JavaScript Baseline Compiler

The SpiderMonkey Baseline Compiler is the first tier of the JIT compiler. It produces native code for JavaScript through stub method calls and optimized inline caches (ICs) for some operations. To adapt for changes to the equality (both strict and non-strict) comparison operation, the fallback stub code was modified to do a call to the VM and test for equality between the two objects in exactly the same manner as in the interpreter.

For the *isTransparent* trap implementation we stop emitting the optimized ICs for object-object comparison, and instead use the fallback IC to do a VM call. This will invoke the *isTransparent* trap for the proxy and then compare with the identity object if it evaluates to *true* or it performs the standard object-object comparison.

For the *TransparentProxy* implementation, we stop emitting the optimized IC for any object-object comparison that involves comparison of *TransparentProxy* object. We use the fallback stub to do a VM call and do a comparison with the identity object, if it involves a *TransparentProxy*. Any other kind of comparison operations are left unaffected and still take place through the optimized stubs.

6.4 IonMonkey Compiler

The IonMonkey optimizing compiler is the final tier of the SpiderMonkey JIT compiler. This has been left unmodified, but we give an outline for a future implementation so that generated native code by IonMonkey can support transparent proxy comparisons.

For the *isTransparent* trap implementation, it will need to load the object into the register and test if the object's class is a *Proxy* or not. If it is not a proxy, then normal fast path for object-object comparison code can be generated. Otherwise execution should stop, do a VM call to the *isTransparent* trap in the handler object of the proxy, and store the return value in a register. If the value is *false* then proceed with emitting the usual object comparison operation code, otherwise load the identity object of the proxy (i.e., the result of calling

GetIdentityObject on the proxy) and replace that value in the equality operation's operand register. Then we can proceed with emitting of code for a standard object comparison.

For the *TransparentProxy* implementation, the code generation for the equality operation would be simpler, as there is no handler trap to be called. We'd have to load the object into the register and test if the object's class is a *TransparentProxy* or not. If it is not a *TransparentProxy*, then normal fast path for object-object comparison code can be generated. Otherwise load the result of calling *GetIdentityObject* on the proxy and replace that value in the equality operation's operand register. Then we can proceed with emitting of code for a standard object comparison.

6.5 Getting the Source Code

The implementation of both modified engines is available on the Web[11]. The branch *isTransparent-trap*[12] contains the isTransparent handler trap version and branch *global-tproxy-object*[13] contains the implementation of the TransparentProxy object. A *README* file in each of the respective branches contains the build instructions.

7 Evaluation

This section reports our experiences with applying our modified engines to JavaScript benchmark programs to answer the following research questions:

RQ1 Does the introduction of transparent proxies affect the performance of non-proxy code?

RQ2 Does a contract implementation based on opaque proxies affect the meaning of realistic programs?

7.1 Performance Test

To answer **RQ1**, we evaluate the impact of our implementations of transparent proxies on programs that do not make use of proxies at all. These programs may be affected by our modifications to the equality comparison algorithms and to the set and map abstraction.

To this end we used benchmark programs from the Google Octane 2.0 Benchmark Suite[14]. This suite comprises 17 programs that range from performance tests to real-world web applications (Figure 3), that is, 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.

Octane reports its results in terms of a score. The Octane FAQ[15] 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.

[11] `https://github.com/sankha93/js-tproxy/`
[12] `https://github.com/sankha93/js-tproxy/tree/isTransparent-trap`
[13] `https://github.com/sankha93/js-tproxy/tree/global-tproxy-object`
[14] `https://developers.google.com/octane/`
[15] `https://developers.google.com/octane/faq`

Benchmark	Origin		Trap		Transparent	
	No-Ion	No-JIT	No-Ion	No-JIT	No-Ion	No-JIT
Richards	505	64.8	502	63.3	509	64.3
DeltaBlue	453	82.5	466	80.2	466	79.6
Crypto	817	111	825	113	793	109
RayTrace	462	182	455	173	462	174
EarleyBoyer	909	275	938	271	913	270
RegExp	853	371	842	362	871	365
Splay	802	409	792	398	857	409
SplayLatency	1172	1336	1222	1307	1231	1338
NavierStokes	841	155	836	156	834	148
pdf.js	2759	704	2764	697	2793	691
Mandreel	691	82.5	711	82.4	688	78.5
MandreelLatency	3803	526	3829	514	3829	503
Gameboy Emulator	4275	556	4250	535	4382	540
Code loading	9063	9439	9124	9318	9114	9502
Box2DWeb	1726	289	1750	278	1736	282
zlib	28981	29052	29097	29074	28909	29108
TypeScript	3708	1241	3715	1210	3666	1203
Total Score	1594	456	1604	447	1610	445

Figure 3 Scores for the Google Octane 2.0 Benchmark Suite (bigger is better). Column **Origin** gives the baseline scores for the unmodified engine. Column **Trap** shows the score values of the engine that implements the transparency trap and column **Transparent** contains the scores for running the engine containing the transparent proxies. The sub-column **No-Ion** (*no IonMonkey*) lists the scores with the baseline compiler enabled, but with IonMonkey disabled. Sub-column **No-JIT** (*no just in-time compilation*) shows the scores of the interpreter without any kind of just in-time compilation.

7.1.1 The Testing Procedure

All benchmarks were run on a machine with two AMD Opteron processors with 2.20 GHz and 64 GB memory. All measurements reported in this paper were obtained with *SpiderMonkey JavaScript-C24.2.0*.

To evaluate our implementation we run the benchmark program in different settings to separate the impact caused by the interpreter and the baseline compiler. Recall that no modifications were done to IonMonkey. Enabling IonMonkey in this state would lead to meaningless results from executing a mixture of proxy-aware code and proxy-oblivious code. Therefore, **the IonMonkey compiler remains disabled**.

7.1.2 Results

Figure 3 contains the score values of all benchmark programs in different configurations explained by the figure's caption. The examples show the run-time impact of our modified engines vs. an unmodified engine. All scores were taken from a deterministic run, which requires a predefined number of iterations, and by using a warm-up run.

Comparing the total scores of the interpreter (column **No-JIT**), the **Trap** version is 1.97% slower than the unmodified engine and the **Transparent** version is 2.41% slower than the unmodified engine. However, when comparing the total scores of the baseline compiler (column **No-Ion**) we see that the **Trap** version is 0.62% faster than the unmodified engine and that the **Transparent** version is 1.00% faster than the unmodified engine.

At this point we have to mention that both differences are smaller than the standard deviation of the mean total scores produced by an unmodified engine. When measuring five runs with the same configuration we found a standard deviation of 2.68 score points for the interpreter and a standard deviation of 23.44 points for the baseline compiler.

The numbers clearly show that both implementations do not have a statistically relevant impact on the execution time of non-proxy code. This result is not surprising because the overwhelming majority of equality comparisons have at least one non-object operand. As our modification only applies when both operands are objects it is only exercised rarely (cf. Section 6) and hence its performance impact is not measurable.

7.2 An Analysis of Object Comparisons

In this section, we answer **RQ2**, by considering how many object-object comparisons occur during a normal program execution and how many of these may fail when objects were wrapped by contracts implemented using opaque proxies. To this end, we count object comparisons involving JavaScript Proxies and give a classification that covers different types of object comparisons whose result might be influenced by the transparency of the involved proxy objects.

7.2.1 The Testing Procedure

For this experiment, we instrumented the JavaScript engine with a monitor to count and classify object-object comparisons. Our subject programs are again taken from the Google Octane 2.0 Benchmark Suite. Our wrapper model is a simple contract system which applies a dynamic type check (cf. Section 2.3) to the arguments of selected functions.

To prepare for the experiment, a source-to-source translation generates for each function that occurs in a program a new variant of the program, where exactly this function is replaced by a function that wraps its arguments with a transparent proxy. The proxy's handler implements a membrane. It forwards the operation to the target and wraps its return value in another transparent proxy. We rely on a weak map to avoid creating chains of nested proxies.

We applied this translation to the benchmark programs and executed each variant in our new engine, counting and classifying each object comparison. As we used transparent proxies, normal execution of the programs is not influenced. Because each function is wrapped individually, we can accurately detect the effect of each single placement of a contract.

7.2.2 Results

First we introduce the types of object comparisons that must be distinguished. We only consider comparisons between two objects where at least one of them is a proxy object, because these are the only comparisons that may be affected if the proxy is opaque.

Type-I All comparisons between a proxy object and another object, which is either a native object or a proxy object from another membrane. Comparisons of this type always return *false* when using opaque proxies. With transparent proxies the result is either *true* or *false*, depending on the proxy's target object.

Type-Ia The subset of **Type-I**, where the underlying target objects differ. Opaque and transparent proxies yield the same outcome, *false*, but for different reasons.

Table 1 Number of comparisons involving object proxies. Column **Total** contains the total number of comparisons. Column **Type-I** lists the comparisons of **Type-I**, divided in the two categories **Type-Ia** and **Type-Ib**. Column **Type-II** shows the number of **Type-II** comparisons, divided in the categories **Type-IIa** and **Type-IIb**.

Benchmark		Type-I		Type-II	
	Total	Type-Ia	Type-Ib	Type-IIa	Type-IIb
DeltaBlue	144126	29228	1411	33789	79698
RayTrace	1075606	0	0	722703	352903
EarleyBoyer	87211	8651	6303	53389	18868
TypeScript	801436	599894	151297	20500	29745

Type-Ib The subset of **Type-I**, where the underlying target objects are the same. A comparison of this type yields *false* when using opaque proxies, whereas transparent proxies yield *true*.

Type-II All comparisons between two proxy objects from the same membrane. If using an identity preserving membrane, a target object is only wrapped once. In this case the result will be *true* if and only if they refer to the same target object, independent of the transparency of the proxies involved. Without an identity preserving membrane, the use of opaque proxies yields *false*, whereas the result with transparent proxies depends on the proxy's target object.

Type-IIa The subset of **Type-II**, where the target objects differ. Opaque and transparent proxies yield the same outcome, *false*, but for different reasons.

Type-IIb The subset of **Type-II**, where both proxies refer to the same target object. A comparison of this type yields *false* when using opaque proxies without an identity preserving membrane, whereas transparent proxies or an identity preserving membrane yield *true*.

In this setting we count the comparison between two proxy objects from different membranes in category **Type-I**, because different contracts implement different membranes and the mechanism that preserves the identity does not work when using different membranes.

Clearly, the **Type-Ib** comparisons are the bad guys as they may flip. They are closely followed by **Type-IIb** comparisons, although they are avoidable if identity preserving membranes are used throughout.

Table 1 summarizes the number of comparisons between native objects and proxy objects and among proxy objects. Comparisons between two primitive values (e.g., a boolean, a number, a string, null, or undefined), comparisons between a primitive value and an object (proxy or native object), and comparisons between two native objects are omitted from the results because the result of the operation is not influenced by the transparency of proxy objects.

Benchmark programs not listed in this table do not contain comparisons with a proxy object: any function in the unlisted programs may be wrapped using any kind of membrane without affecting its meaning. However, they still contain comparisons between native objects and primitive values (usually the test $ptr === null$).

The numbers in the table cover all comparison operators, namely *equal* ($==$), *not equal* ($!=$), *strict equal* ($===$), and *strict not equal* ($!==$). The meaning of "the result is *true*" is generalized to the sense that *equal* and *strict equal* will return *true*, *not equal* and *strict not equal* will return *false*.

What we see in Table 1 is that there are three benchmarks with a non-negligible number of bad **Type-Ib** comparisons, although the majority of object comparisons is not affected. The numbers also indicate that there are many more **Type-IIb** comparisons, so that any use of non-identity preserving membranes should be strongly discouraged.

7.3 Summary and Threats to Validity

The evaluation shows that the implementation of a dynamic contract system based on opaque proxies, whose monitoring replaces function arguments by proxy objects, definitely influences the program execution (which answers **RQ2**), which in turn leads to program errors.

The reason for the comparatively small number of flipped comparisons is due to the careful handling of object comparisons in JavaScript. Results from previous unpublished experiments show that approximately 6% of all comparisons involve two objects. The vast majority of comparisons either check an object against null or undefined, or compare primitive values.

8 Related Work

The JavaScript proxy API [20, 21] enables a developer to enhance the functionality of objects easily. The implementation of proxies opens up the means to fully interpose all operations on objects including functions calls on function objects.

JavaScript proxies have been used for Disney's JavaScript contract system, contracts.js [5], to enforce access permission contracts [13], as well as for other dynamic effects systems, meta-level extension, behavioral reflection, security, or concurrency control [16, 2, 4].

Proxy-based implementations avoid the shortcomings of static implementations and offline code transformations. In JavaScript, static approaches are often lacking because of the dynamicity of the language. Proxies guarantee full interposition and handle the full JavaScript language, including the *with*-statement, *eval*, and arbitrary dynamic code loading techniques.

The ideal contract system should not interfere with the normal execution of code as long as the application code does not violate any contract. The application should run as if no contracts were present [7].

Object equality becomes an issue for non-interference when contracts are implemented by some kind of wrapper. The problem arises if an equality test between wrapper and target or between different wrappers for the same target returns false instead of true. This issue is known from other work involving wrappers for implementing object extensions and multimethods [22, 3]

The PLT group examines various designs for low-level mechanisms for implementing contracts and related abstractions [19]. They propose two kinds of proxies, chaperones and impersonators, that differ, for example, in the degree of freedom for modifying the underlying object. They experience similar problems with noninterference as we report in Section 3.3 when using the *eq?* operator which is similar to JavaScript's strict equality operator `===` and roughly implements pointer equality on objects. Racket's chaperones and impersonators are not transparent with respect to this operator. However, the preferred equality operation in Racket, *equal?*, implements structural equality which is indifferent to proxy transparency. In contrast, JavaScript provides no built-in operation to test for structural equality, so that developers need to build on pointer equality or roll their own structural equality.

9 Conclusion

Neither the transparent nor the opaque implementation of proxies is appropriate for all use cases. We discuss several amendments and propose two flexible solutions that enable applications requiring transparency as well as opacity. Both solutions are implemented as extensions of the SpiderMonkey JavaScript VM. This approach ensures full and transparent operation with all JavaScript programs. Hence, we can evaluate the solution on real-world JavaScript programs.

A significant number of object comparisons would fail when mixing opaque proxies and their target objects. This situation can arise when gradually adding contracts to a program during debugging. Identity preserving membranes decrease this number, but they are not able to guarantee full noninterference.

We also measured the run-time impact of an implementation supporting transparent proxies on the execution time of a realistic program mix. The results show that the modification to equality required to support transparent proxies has no statistically significant impact on the execution time.

References

1 Michael D. Adams and R. Kent Dybvig. Efficient nondestructive equality checking for trees and graphs. In Peter Thiemann, editor, *Proceedings International Conference on Functional Programming 2008*, pages 179–188, Victoria, BC, Canada, September 2008. ACM Press, New York.

2 Thomas H. Austin, Tim Disney, and Cormac Flanagan. Virtual values for language extension. In Cristina Videira Lopes and Kathleen Fisher, editors, *OOPSLA*, pages 921–938, Portland, OR, USA, 2011. ACM.

3 Gerald Baumgartner, Martin Jansche, and Konstantin Läufer. Half & half: Multiple dispatch and retroactive abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Revised 3/02, Ohio State University, March 2002.

4 Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA*, pages 331–344. ACM, 2004.

5 Tim Disney. contracts.js. `https://github.com/disnet/contracts.js`, April 2013.

6 Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten your JavaScript: Hygienic macros for ES5. In Andrew P. Black and Laurence Tratt, editors, *DLS*, pages 35–44, Portland, OR, USA, October 2014. ACM.

7 Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. In Olivier Danvy, editor, *Proceedings International Conference on Functional Programming 2011*, pages 176–188, Tokyo, Japan, September 2011. ACM Press, New York.

8 ECMAScript Language Specification, December 2009. ECMA International, ECMA-262, 5th edition.

9 ECMAScript Language Specification. `http://wiki.ecmascript.org/lib/exe/fetch.php?id=harmony:specification_drafts&cache=cache&media=harmony:ecma-262_6th_edition_final_draft_-04-14-15.pdf`, April 2015. ECMA International, ECMA-262, 6th edition (draft).

10 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In Simon Peyton-Jones, editor, *Proceedings International Conference on Functional Programming 2002*, pages 48–59, Pittsburgh, PA, USA, October 2002. ACM Press, New York.

11 Matthew Flatt, Robert Bruce Findler, and PLT. *The Racket Guide*, v.6.0 edition, March 2014. `http://docs.racket-lang.org/guide/index.html`.

12 Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent object proxies for JavaScript. Technical report, Institute for Computer Science, University of Freiburg, 2015.

13 Matthias Keil and Peter 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.

14 Matthias Keil and Peter Thiemann. TreatJS: Higher-order contracts for JavaScript. `http://proglang.informatik.uni-freiburg.de/treatjs/`, 2014.

15 Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

16 Mark S. Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in JavaScript. In Matthias Felleisen and Philippa Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 1–20, Rome, Italy, March 2013. Springer.

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

18 Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten, editors. *Revised[6] Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.

19 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In Gary T. Leavens and Matthew B. Dwyer, editors, *OOPSLA*, pages 943–962. ACM, 2012.

20 Tom Van Cutsem and Mark S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In William D. Clinger, editor, *DLS*, pages 59–72. ACM, 2010.

21 Tom Van Cutsem and Mark S. Miller. Trustworthy proxies – virtualizing objects with invariants. In Giuseppe Castagna, editor, *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 154–178, Montpellier, France, July 2013. Springer.

22 Alessandro Warth, Milan Stanojevic, and Todd 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.