

# Caja

## Safe active content in sanitized Javascript

Mark S. Miller    Mike Samuel    Ben Laurie    Ihab Awad    Mike Stay

October 31, 2007

### Abstract

Using Caja, web apps can safely allow scripts in third party content.

The computer industry has only one significant success enabling documents to carry active content safely: scripts in web pages. Normal users regularly browse untrusted sites with Javascript turned on. Modulo browser bugs and phishing, they mostly remain safe. But even though web apps build on this success, they fail to provide its power. Web apps generally remove scripts from third party content, reducing content to passive data. Examples include webmail, groups, blogs, chat, docs and spreadsheets, wikis, and more; whether from Google, Yahoo, Microsoft, HP, Wikipedia, or others.

Were scripts in an object-capability language, web apps could provide active content safely, simply, and flexibly. Surprisingly, this is possible within existing web standards. Caja represents our discovery that a subset of Javascript is an object-capability language.

## 1 Introduction

In a memory-safe object language, such as Javascript, object A can only invoke object B if A has a reference to B. If A already has references to B and C, A can invoke B passing C as an argument, giving B access to C. Memory-safe object language with encapsulation, such as Java, *protect objects from their outside world*. The clients of an encapsulated object can make requests using its public interface; but how the object reacts to these requests is up to the object.

An encapsulated object can ensure that causality can only enter it according to the object model of computation.

An *object-capability language* is essentially a memory-safe object language with encapsulation, with additional restrictions that *protect the outside world from the objects*. In an object-capability language, an object can only cause effects outside itself by using the references it holds to other objects. Objects have no powerful references by default, and are granted new references only by normal message passing rules. Object references thereby become the sole representation of rights to effect the world, and normal message passing is the only rights transfer mechanism. An object can be denied authority simply by not giving it those references which would provide that authority.

The browser sandbox already mostly protects the outside world from scripts running on web pages. A great virtue of Javascript is that many people successfully program in it casually, without first learning the language in any depth. Caja<sup>1</sup> is an enforced subset of Javascript we designed to make as little impact as possible on regular Javascript programming, while still providing object-capability security. In this section, we provide a brief inaccurate overview of the differences between Caja and Javascript suitable for the casual Javascript programmer. The rest of this document then accurately goes into more depth.

**Forbidden names.** Caja rejects all names ending with “`--`” (double underscore). This gives the

---

<sup>1</sup> *Caja* is Spanish for “box”. With Caja, capabilities attenuate Javascript authority.

Caja implementation a place to store its book-keeping information where it is invisible to the Caja programmer.

**Frozen objects.** If an object is frozen, an attempt to set, add, or delete its properties will throw an exception instead. Functions and prototypes are implicitly frozen. In addition, the Caja programmer can explicitly freeze objects to prevent their direct modification. All objects in the default global scope are *immutable*, or transitively frozen.

**No shared global scope.** Each separately loaded module has its own global scope which inherits from the default global scope, isolating them from each other.

**Private names.** Property names ending in “\_” (single underscore) serve as protected instance variables. Such names can only appear to the left of “**this**.”. As with Smalltalk instance variables or **protected** instance variables in C++, these protected instance variables are visible up and down the inheritance chain within an object, but are not visible outside an object.

**No method stealing.** The single underscore rule above only protects an object’s state from its clients if its clients cannot add methods to it which alias its “**this**”. Caja divides functions into three categories: *simple functions* are those which do not mention “**this**”. They are first class and can be used without further restriction. *Constructors* are named functions which mention “**this**”. *Methods* are anonymous function which mention “**this**”.

Neither constructors nor methods are first class. They can be used in ways that support stereotyped class-like Javascript programming patterns, but they cannot be stored in variables or passed as arguments. Fortunately, there’s no loss of generality. The Caja programmer can obtain the same effect by wrapping a constructor or a method in a simple function (such as **F.make** in Figure 1) and passing that simple function instead.

```
function F(x) { this.x_ = x; }
F.prototype.getX = function() {
  return this.x_;
};
F.make = function(x) {
  return new F(x);
};
function test() {
  return new F(3).getX() === 3;
}
```

Figure 1: Caja Functions. **F** is a *constructor*. It can only be initialized and used with **new** and **instanceof**. **F.prototype.getX** is a *method*. It can only be called as a method. **F.make** and **test** are *simple functions*. They can be passed around freely.

**Crazy things are gone.** Caja contains no “**with**” or “**eval**”. Caja bundles in a safe JSON library to support the most common use of **eval**.

Hopefully, this is all the casual Caja programmer needs to know to get started. Section 2 explains a bit of the history of access control in the web browser, in order to motivate the problems Caja addresses. It may safely be skipped. Section 3 explains the problems faced when securing Javascript, many of which involve the use of “**this**”.

We then present Caja in two stages. Section 4 presents *Cajita*, the subset of Caja without “**this**”. For new code, Cajita is a reasonably expressive language resembling an object-oriented Scheme. Section 5 then presents the remainder of the Caja language beyond Cajita. Caja adds back enough of Javascript for most old habits and old code to port pleasantly and painlessly. Caja and Cajita interoperate without problems. Section 6 briefly surveys related work.

## 2 Identity-centric Epicycles

When a document contains live interactive programs, we say it contains *active content*. The computer industry has spent over a billion dollars in failed at-

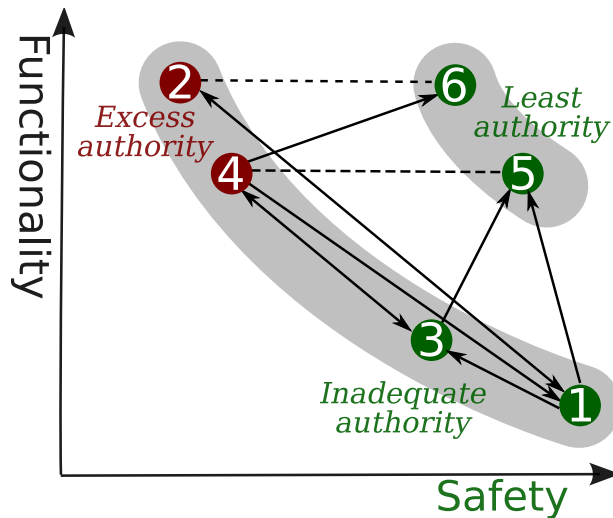


Figure 2: The Evolving Authority of Active Content. Identity-based access controls have led to thrashing between lost functionality and lost safety. To have both, we need to provide *least authority*: adequate authority for desired functionality without excess authority which invites abuse.

tempts to support active content. But the success of web apps—themselves a form of active content—demonstrates that this dream was worth pursuing. Unfortunately, web developers today face a maze of complex security mechanisms that have, so far, prevented web apps themselves from supporting active content. To navigate our way out of this maze, we must first see how we got here.

Today’s desktop operating systems all use some form of identity-based access control [5], in which an installed application runs *as* its user, and so is entrusted with all its user’s authority. Such an application can provide its user all the functionality modern operating systems support, but at the price of being able to do anything its user may do. We depict this situation at 2 on Figure 2. When you run Solitaire, it can delete all your files while playing within the rules of your system, without exploiting any bugs. (For the remainder of this document, we will ignore hazards due to implementation bugs, and explain only hazards due to architectural choices.)

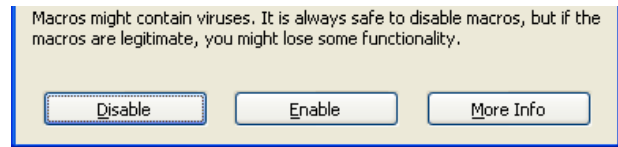


Figure 3: Only Bad Choices. When documents contain scripts, users can disable themselves from getting any work done 1 or enable scripts to destroy all their other work 2.

At first, the documents handled by applications were safe passive data 1. Applications first supported active content by running scripts in documents with all of their user’s authority 1→2. Excess authority invites abuse. Simply “reading” a malicious document would allow it to delete all your files. In reaction, installed office applications now encourage users to disable scripts (Figure 3) reducing content back to passive data 2→1. The failures of excess authority shown on the upper left thus led to the failures of inadequate authority shown on the lower right.

The web browser is itself an installed application that runs scripts in two contexts. Browser extensions run with all the user’s authority 2. Scripts in web pages run sandboxed, with no authority to the user’s local files. The browser’s *same origin policy*, another layer of identity-based control [14], provides scripts with the authority to communicate with their site of origin 1→3. Regarding both decisions, the user is helpless. The user has no practical way to grant a script the authority to edit one of the user’s local files, nor can the user deny a script the ability to call home. So long as the user’s valuable assets were local, this model successfully protected the user.

Web apps leverage this success. To the browser, the page on which a web app resides is a document, and the web app itself is simply active content within that document. But to the user, a web app is an application managing yet other documents on the user’s behalf. For example, when the user interacts with webmail, the documents of interest are email messages. Likewise for groups, blogs, chat, docs and spreadsheets, wikis, and more. Let us refer to the documents managed by web apps as *passages*, to dis-

tinguish them from the web pages on which they appear.

Since the user can neither grant a web app access to local files nor deny it the ability to call home, the only place a web app could store these passages is on its site of origin. The browser security model protected the user’s local files from being harmed *or used*. As users shift to using web apps, the assets they value come to be the passages stored at these various origin sites.

To protect their user’s remote passages, web apps employed yet another layer of identity-based controls, relying on cookies or other forms of authentication to identify their user. But when scripts within these passages ran, they would run within the web page containing the web app serving them, and were thereby authorized to do anything their web app could do on behalf of its user ④. For example, if a webmail application allowed HTML email messages to carry scripts, simply “reading” an incoming email message would allow it to delete your inbox. The ③→④ transition is not a technical change, but a change in where the user’s value resides, and thus a change in the user’s risks. By this dynamic, failures of inadequate authority led to failures of excess authority.

To protect against malicious passages, some web apps do safely provide active content using *iframes*—effectively nested web pages—at the cost of isolating themselves from this content ④→③ [14]. Most web apps *sanitize* HTML content by removing all scripts, reducing content again to passive data ④→①. Existing HTML sanitizers disinfect the patient but leave a corpse. This recapitulates the loss of active content in installed office applications. Some proposals would address these next incremental problems by adding yet another identity-centric epicycle. Can we do better?

If we could start over again, we could use an authorization-centric model such as object-capabilities [2]. The object-capability alternative naturally supports POLA, the principle of least authority, shown in the upper right. An object in an object-capability language can only cause effects by invoking the public interfaces of objects it can reach. An invocation provides references to other objects as

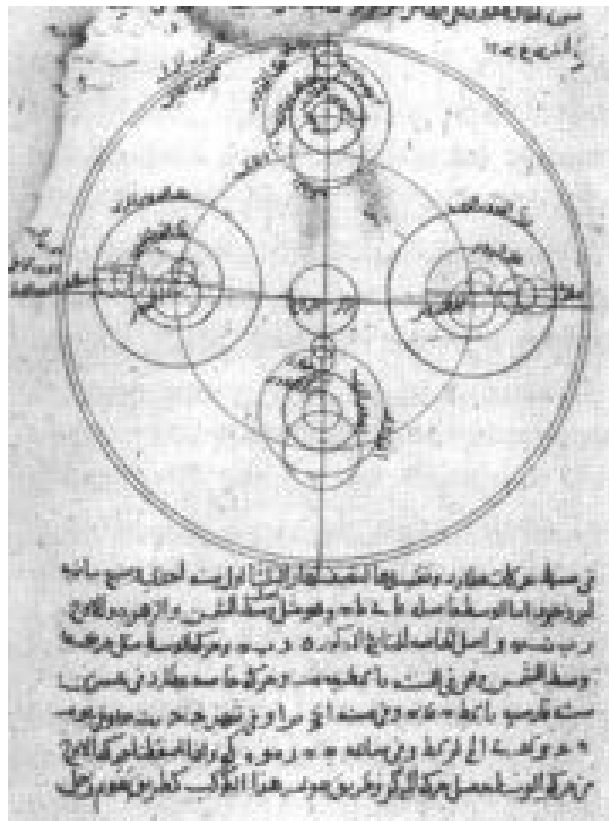


Figure 4: Ptolemy’s epicycles. Ptolemy attempted to model the motion of the heavenly bodies using only circles. With each discovery that the model didn’t fit, yet another layer of circle was added to adjust. By contrast, Kepler’s ellipses fit the problem directly, with no need for endless additional layers.

arguments, providing the invoked object the least authority needed to carry out these requests [8]. Within these rules, active content would run with exactly the authority explicitly provided by its containing document. Surprisingly, we can gain these benefits simply by applying a milder, non-lethal sanitizer.

Experience with Java, Scheme, OCaml, Pict, and others demonstrates that existing memory safe languages often already contain an expressive object-capability subset [7, 9, 11, 6, respectively]. We refer to the object-capability subset of Javascript as *Caja*.

The Caja translator restricts the Javascript it accepts to be within this subset. This sanitized Javascript is still a general purpose object programming language which Javascript programmers should find familiar, pleasant, expressive, and easy to learn and use.

Some web apps could use the Caja translator to allow active content in their passages ①→⑤. Other web apps could use Caja to overcome the limits of iframes ③→⑤. Browser extensions, which run with their user’s full authority, could make a *powerbox* available to scripts in pages [13, 12, 10, 4]. A web app, on detecting the presence of a powerbox, could offer to edit a local file chosen by the user ④→⑥.

Our starting point is Javascript 1.5 as documented in the third edition of the EcmaScript 262 standard [3]; hereafter *ES3*. The remainder of this document explains the differences between Caja—the Javascript subset accepted by the Caja translator—and ES3. Other documents will explain Caja’s sanitization of the remaining elements of active web content: HTML, CSS, and the DOM and other APIs provided by browsers to Javascript. We refer collectively to the subset of these accepted by the Caja translator as *Caja web content*.

### 3 Subsetting Javascript

Caja defines a subset of Javascript both syntactically and semantically. The Caja translator rejects non-Caja input statically when it can. But because of Javascript’s dynamic nature, some of Caja’s restrictions cannot be imposed statically, so the Caja translator translates the Javascript it accepts into Javascript with additional runtime checks. To facilitate development, it is easy to write a Caja program so it can run correctly whether it is run as a Caja program or run directly as an untranslated Javascript program.

A web app (or any other Javascript-based embedding application framework) can be written partially in Javascript and partially in Caja. The web app loads the Caja runtime library, which is written in Javascript, and which is assumed by the Javascript output of the Caja translator. All untrusted scripts must be provided as Caja source code, to be verified

and translated by the Caja translator. The translator’s output is either included directly in the containing web page or loaded by the Caja runtime.

A loose analogy with machine and operating system architecture should help explain the relationships. In the analogy, the full Javascript language serves the role of the machine’s full instruction set. Javascript’s global scope serves the role of physical memory addresses. The I/O-capable objects provided to Javascript by a hosting environment, such as the DOM objects provided by the browser, serve the role of devices.

**User-mode.** By a combination of static and dynamic checks, the translator allows only a safe “user-mode” subset of Javascript. As with user-mode instructions, this subset can compute any computable function, but cannot cause external effects nor sense the outside world.

**Address mapping.** A package of Caja source code to be translated together defines a *Caja module*. All code within the same module share a global scope, but distinct modules see disjoint global scopes. The translator maps Caja global variable references to instead address module-relative fields.

**Context switching.** When Caja object A has a reference to Caja object B, this should enable A to invoke B’s public interface but not access B’s internal state. A and B should both be able to defend their integrity from the other’s possible misbehavior.

**System calls, device drivers.** When a Caja object A invokes an object B written directly in Javascript, the operations provided by B serve the role of system calls. Caja protects B from A, but A is fully vulnerable to B. When B is a safe wrapper around one of the host’s device-like objects, such as a DOM node, B also serves as a device driver.

A “system call” corresponds to a Caja object invoking a Javascript object. A web app that is written entirely in Javascript and provides many services to

its Caja objects directly would be like a monolithic kernel. For compatibility with existing Javascript apps, we support this usage pattern but we don't recommend it. By analogy with kernel code at the boundary with untrusted code, such Javascript code needs to maintain delicate invariants that it is easy to get wrong.

The other extreme is analogous to a micro-kernel. The minimal necessary Javascript code would be the app-neutral Caja runtime itself, and a small app-dependent powerbox providing device drivers and initialization. All other services should be Caja objects to be invoked by other Caja objects. Most of the logic of a web app should be structured as such Caja-based services.

### 3.1 Javascript specific problems

Most of the above remarks would apply equally well were we starting from various other base languages. There are additional issues peculiar to Javascript that we must deal with. Many of these issues are also software engineering hazards for which Javascript programmers have developed defensive programming conventions. Where possible, Caja copes with these issues by adapting and enforcing these existing conventions.

**Unconstrained Properties.** Javascript objects contain *properties*, i.e., named fields holding references to other objects. Javascript specifies that some properties are constrained to be *Internal*, *ReadOnly*, *DontEnum*, or *DontDelete*. Such constraints would help an object protect itself from its clients, but Javascript provides no way to express these constraints in the language. Instead, any object defined in Javascript is freely mutated by any other object with access to it.

**Global scope.** All Javascript code executing within the same Javascript engine (such as a web page or iframe) implicitly share access to the same global scope. Therefore, in Javascript, objects cannot be isolated from each other.

**Implicit mutable state.** Some base Javascript objects, such as `Array.prototype`, are implicitly

reachable even without naming any global variable names. Even after global scope problems are fixed, the mutability of these objects would prevent isolation.

**Lack of encapsulation.** To support the “context switching” criterion explained above, objects need to be able to encapsulate their private state. Javascript does provide one such mechanism: lexical variables captured by lexical closures. However, using this as the sole encapsulation mechanism for object patterns conflicts with existing Javascript programming practice.

**“this” what?** Javascript’s rules for binding “this” depend on whether a function is invoked by *construction*, by *method call*, by *function call*, or by *reflection*. If a function written to be called in one way is instead called in another way, its “this” might be rebound to a different object or even to the global scope.

**Foreign for/in loops.** Javascript’s `for/in` loop enumerates the names of all an object’s properties, whether inherited or not, unless the property is `DontEnum`, which the Javascript programmer has no way to express. As a result, some of these names need to be skipped by the loop body. Every Javascript coding style invents its own defensive pattern of additional tests to skip unwanted property names.

**Weak static analysis.** Although Caja is less dynamic than Javascript, we still assume that it is impractical to perform any interesting analysis, such as type inference, both statically and safely. As a result, Caja’s static restrictions can only enforce simple syntactic rules. Remaining restrictions must be enforced by runtime checks.

**Fast path.** For the micro-kernel approach to be attractive, Caja’s extra runtime checks must not cost too much. Frequent operations, such as property access using “.” must run close to full speed.

**Uncontrolled language growth.** The ES3 spec allows one to add new dangerous properties

to core objects while claiming ES3 compatibility. Javascript language implementors, platform providers, and standards committees may use of this freedom with unpredictable results. For example, some Javascript implementations have added dangerous properties, like `eval`, to core objects, like `Object.prototype`. A safe subset must deny access to these additional unknown properties. But since these new properties are often `DontEnum`, there isn't even a reliable way to detect them.

**Browser compatibility.** Web content must work on widely deployed browsers whether on not these browsers strictly conform to the relevant standards. At the time of this writing, the plausible baseline platform is the intersection of ES3, Firefox 1.5, Internet Explorer 6, Opera 8.5, Safari 3, and their successors. Fortunately, these browsers do conform quite closely to ES3. Later versions of Caja may specify larger subsets of EC3.

**Silent errors.** In Javascript, various operations, such as setting a `ReadOnly` property, fail silently rather than throwing an error. Program logic then proceeds along normal control flow paths premised on the assumption that these operations succeeded, leading to inconsistency. To program defensively in the face of this hazard, every assignment would be followed by a “*did it really happen?*” test. This would render programs unreadable and unmaintainable. Where practical, Caja deviates from standard Javascript by throwing an exception rather than failing silently.

**Feature testing.** The Javascript *feature testing* pattern relies on a failed attempt to read a property returning `undefined` rather than throwing an exception. Since, in this case, the program naturally notices the problem anyway, Caja does not turn this case into a thrown exception.

This last point about “Silent errors” is another reason to avoid the monolithic kernel approach. Web apps in untranslated Javascript are vulnerable to any

```
function Counter() {
  var count = 0;
  return caja.freeze({
    toString: function() {
      return "<counter: " + count + ">";
    },
    incr: function() {
      return count += 1;
    },
  });
};
```

Figure 5: A Cajita Counter. Each call to `Counter()` or produces a new counter object. Access to a counter provides the authority to read, invoke, or enumerate its properties, all of which are simple functions serving the role of methods. Caja functions are implicitly Frozen; the returned object is explicitly Frozen; and the instance-state of the object—the count variable—is accessible only as encapsulated state captured by these pseudo-methods. As a result, a reference to a counter object is a proper protected capability, providing least authority to a shareable counter service.

malicious active content that finds a way to provoke a silent error and exploit the resulting inconsistency.

Caja is only a subset of Javascript in a limited sense: While a Caja program has not explicitly indicated a failure, it executes within Javascript's semantics. By *indicate a failure*, we mean either throwing an exception or returning `undefined` for a property read.

## 4 Cajita Specification

Most of the complexity of Caja is needed to defend against Javascript's rules regarding the binding of “`this`”. The subset of Caja without “`this`” is a perfectly reasonable and expressive programming language. Caja supports “`this`” in order to ease the porting of old code. For new code, we recommend sticking to the thisless subset of Caja, which we refer to as *Cajita*, the diminutive, meaning “small box”. (The Caja translator may eventually provide a switch

```

function Point(x, y) {
  return caja.freeze({
    toString: function() {
      return "<" + x + "," + y + ">";
    },
    getX: function() { return x; },
    getY: function() { return y; },
  });
};

var pt = Point(3, 5);

```

Figure 6: A Cajita Point. As a baseline, we first express this simple example in Cajita with no support for inheritance. Other elaborations will show how to support inheritance and various styles of definition in both Cajita and full Caja.

```

function PointMixin(self, x, y) {
  self.toString = function() {
    return "<" + self.getX() + "," +
      self.getY() + ">";
  };
  self.getX = function() { return x; };
  self.getY = function() { return y; };
  return self;
}
function Point(x, y) {
  return caja.freeze(PointMixin({}, x, y));
}

```

Figure 7: Cajita Inheritance. In the Cajita inheritance pattern, the equivalent of a non-final class is a function ending with “`Mixin`” with `self` as its first parameter. The method-like functions can use `self` analogously to the use of `this` in full Caja, in order to refer to the overall object being defined. This “`*Mixin`” function should only be called by “sub-classes” such as `WobblyPointMixin` below. If the class is non-abstract, it should also have a pseudo-constructor function such as `Point` for making direct instances.

```

function WobblyPointMixin(self, x, y) {
  var super = caja.snapshot(self);
  self.getX = function() {
    return super.getX() + Math.random();
  };
  return self;
}
function WobblyPoint(x, y) {
  var self = PointMixin({}, x, y);
  self = WobblyPointMixin(self, x, y);
  return caja.freeze(self);
}

```

Figure 8: Cajita `WobblyPointMixin`. The equivalent of a non-final subclass is a “`*Mixin`” function with `self` as its first parameter, where the body calls `caja.snapshot` to make a frozen copy of the partially initialized `self` at that moment, to serve as the conventional `super` for the other functions defined within this scope.

to enforce that a translated program is within Cajita.)

TODO To be written

## 4.1 Definitions

To explain the restrictions Cajita imposes, we need some definitions.

**JSON Container.** An object whose prototype’s “`constructor`” property is `Array` or `Object`, i.e., under normal conditions, an object inheriting directly from `Array.prototype` or `Object.prototype`. JSON containers are normally created using the `[...]` or `{...}` syntaxes.

**Simple functions.** A function whose body does not mention “`this`” is a *simple function*. A simple function can be either named or anonymous. A simple function can be invoked *as a function* (`foo(a...)`), *as a method* (`foo.id(a...)`), or *as a constructor* (`new Foo(a...)`). Simple functions are first class—they can be stored in

variables and passed around freely, just like any other value.

**Frozen.** If an object is *Frozen*, any attempt to directly assign to its properties, add new properties to it, or delete its properties causes an exception to be thrown. Frozen is a shallow restriction: Frozen objects can retain and provide non-Frozen objects. (Imagine a frozen surface covering a liquid lake.) Once initialized, all Caja functions—simple functions, constructors, and methods—are implicitly frozen, as are constructor prototypes. The Caja runtime library additionally provides an explicit operation for freezing an object: `“caja.freeze(obj)”`.

**Immutable.** If an object is *Immutable*, then it is Frozen, and all objects it has access to are themselves Immutable. Shared access to an Immutable object does not provide a communication channel, and so does not endanger isolation. With the exception of `Math.random` and `Date`, all objects that are globally or implicitly accessible to all Caja programs are Immutable. We discuss these exceptions below.

## 4.2 Static restrictions

Any source code statically accepted by the Caja translator is *verified Caja program text*. The presence of the following syntactic elements causes an alleged Caja program text to instead be statically rejected.

**Stable language.** Virtually any input which should be statically rejected by ES3 is forbidden, even if it would be allowed by a target browser or later Javascript specifications. This includes any use of keywords reserved in ES3. But we reserve the right to include *de-facto extensions* to ES3 as explained below.

**De-facto extensions.** As we identify widely supported extensions of ES3 that we can accept as input, but still translate to conforming EcmaScript on output, we may add these to Caja. For example, we are currently considering allowing backslash as a line continuation character,

since this is allowed by virtually all Javascript implementations and can be trivially translated to correct ES3.

**Without “with”.** The “with” keyword is forbidden. Because of the scope confusion it causes, “with” is a widely hated and avoided feature that would be a lot of trouble to support safely.

**Beware unicode.** Non-Latin-1 characters are forbidden for now in source text outside of string literals. Some of these create parsing problems on some widely deployed Javascript platforms. Prohibiting these protects against some char-encoding attacks. We expect to relax this restriction once we know how to do so safely.

**Forbidden names.** An identifier ending with a double underscore is forbidden, either as a variable name or a property name. We reserve the triple underscore for use by the Caja translator output and the Caja runtime code. Firefox reserves use of the double underscore for itself.

**RegExp syntax.** In Javascript implementations, the pattern syntax is often optimized into a static object with mutable state, violating isolation. Caja translates the `/pattern/` syntax to `new RegExp("pattern")`.

**Caja features absent from Cajita.** See section 5 for the rules governing the “this” keyword, identifiers ending in a single underscore, and the “new” keyword.

## 4.3 Dynamic restrictions

TODO To be written

## 4.4 Outer scope restrictions

In Javascript, all code loaded into the same Javascript execution environment shares a common mutable global scope. To avoid confusion, we refer to the corresponding concept in Caja as the *outer scope*. The Caja outer scope comes in two layers:

Caja’s *shared outer scope* contains the subset of the standard ES3 global scope that we deem to be

safe. This includes all the objects and properties defined by that standard, with the following caveats. Unless stated otherwise, all the objects in this outer scope are transitively immutable, so that they provide no ambient authority to objects defined within this scope. The shared outer scope itself is frozen.

Each instantiation of a Caja module is a separate plugin. Each plugin executes with a new *per-plugin outer scope* which is mutable and inherits from Caja’s shared outer scope. Therefore, all objects within the same plugin can implicitly communicate and interfere with each other. However, *claim: two separate plugins, even if they instantiate the same module, are isolated from each other.*

**eval** The Caja global scope has no “eval” function. Instead, the “Caja.eval” method will evaluate Caja source code with an explicitly provided set of variable bindings to serve as its initial global scope, as explained below.

**Function** The Javascript `Function` constructor is absent from the default Caja outer scope, and must not be available in Caja.

Unfortunately, `Function` is unsafe even when invoked as a normal function, and we’re currently hoping to stop protecting normal function calls with a runtime check. *Claim: Even without the runtime check on normal function calls, the rest of the restrictions in this document together make `Function` unreachable from Caja programs.*

**function.constructor** The Javascript `function` property of functions is absent from Caja.

**new Date()** In Javascript, “new Date()” gives ambient access to the current date and time, in violation of object-capability rules as well as dependency injection discipline. `Date` is therefore a member of the global scope which is not actually immutable. Further, this ambient access to the current time provides a timing channel, further impeding any attempts to stem the leakage of bits over covert channels. Nevertheless, despite these concerns, because it provides only

a read-only channel for sensing the world, Caja provides the Javascript `Date` constructor to Caja programs.

**Math.random()** The Javascript `Math.random` method is not even read-only. The ES3 standard places no obligations regarding quality of the randomness produced. In particular, an implementation could conform to ES3 and still leak to a given caller of `Math.random()` the ability to infer how many previous times it had been called. Nevertheless, Caja provides the Javascript `Math.random` method to Caja programs. We recommend that Javascript platform providers provide good enough randomness that this method doesn’t serve as an information channel between otherwise-isolated plugins.

## 4.5 The Caja Library

TODO To be written

## 4.6 Examples

# 5 Remaining Caja Specification

TODO To be written

## 5.1 Definitions

To explain the restrictions Caja imposes on Javascript, we need some additional definitions.

**Constructed object.** An object defined by Caja code that’s not a JSON container and not a function must have been constructed by calling “new” on a constructor other than `Array` or `Object`. Constructed objects and the “new” keyword are absent from Caja.

**Constructors.** A named function whose body mentions “this” is a *constructor*. A constructor can only be invoked by construction, i.e., by a call using the “new” keyword. To enforce this, constructors are not *first class values*. A constructor definition and a constructor name can only

```

function F(a...){...}
F.prototype.mname = /*method or function*/;
F.prototype.mname = /*expr with no F*/;
...
F.sname = /*function or expr with no F*/;
...
/*Generate ____freeze(F.prototype); here*/
/*Generate ____freeze(F); here*/
/*first use of F*/

```

Figure 9: Initializing the Default Prototype. The normal syntactic pattern for expressing class-like code is accepted, so long as all behavior definition happens before the first use of `F`. The *mnames* would typically be method names. The *snames* name static members, to be accessed off the constructor.

appear in certain static contexts enumerated below, in order to ensure that the constructor value does not escape. Constructors are absent from Cajita.

**Methods.** An anonymous function whose body mentions “`this`” is a *method*. A method definition may only appear in prototype property initializations. To avoid the confusions regarding “`this`”, Caja methods are also not first class. A property holding a method is *call only*, it can be called but not read. Methods are absent from Cajita.

## 5.2 Static restrictions

Any source code statically accepted by the Caja translator is *verified Caja program text*. The presence of the following syntactic elements causes an alleged Caja program text to instead be statically rejected.

**Internal properties.** An identifier ending in a single underscore may be used only to name Internal properties. It may appear only after a “`this.`”.

Although constructors are normally frozen and the “`prototype`” property of functions is generally not

```

// Shorthand using:
// caja.def(F,Sup,opt_members,opt_statics);

function F(a...){
  F.Super.call(this,b...); //optional
  ...
}
caja.def(F, G, {
  mname: /*method, function, or expr*/),
  ...
}, {
  sname: /*function or expr*/),
  ...
});
/*first use of F*/

```

Figure 10: Shorthand. As a convenience, the translator recognizes this pattern as still defining `F` rather than using it. `caja.def` initializes `F.Super = G`, sets up the normal inheritance relationship between `F` and `G`, and freezes `F.prototype` and `F`.

```

function Point(x, y) {
  this.x_ = x;
  this.y_ = y;
};
Point.prototype.toString = function() {
  return "<" + this.x_ + "," +
    this.y_ + ">";
};
Point.prototype.getX = function() {
  return this.x_;
};
Point.prototype.getY = function() {
  return this.y_;
};
Point.prototype.add = function(other) {
  return new Point(
    this.x_ + other.getX(),
    this.y_ + other.getY());
};

var pt = new Point(3, 5);

```

Figure 11: Generic Point Example.

```

function Point(x, y) {
  this.x_ = x;
  this.y_ = y;
};
Caja.def(Point, Object, {
  toString: function() {
    return "<" + this.x_ + "," +
           this.y_ + ">";
  },
  getX: function() { return this.x_; },
  getY: function() { return this.y_; },
  add: function(other) {
    return new Point(
      this.x_ + other.getX(),
      this.y_ + other.getY());
  }
});
var pt = new Point(3, 5);

```

Figure 12: Brief Point Example.

settable, we allow at least the patterns shown in Figures 9 and 10 for declaring a constructor, initializing it, and initializing its prototype. Figures 11 and 12 show a familiar example expressed in these patterns.

The first argument to “Caja.def” may be the prototype only of a fresh definition of F. None of these initialization may make any other use of F or F.prototype. All initialization of F and its prototype will therefore precede any use of F. This means, in effect, that the assignments above can be considered declarative initializations rather than mutations. The mention of “Point” within the “add” method is not considered a too-early usage of “Point” since it occurs only within a method body that cannot be executed too early.

Claim: *No Caja program can cause an observable mutation of an object Caja considers frozen.*

### 5.3 Dynamic restrictions

TODO To be written

```

function Brand(name) {
  caja.requireType(name, 'string');
  var flag = false;
  var squirrel = null;

  function Sealer(payload) {
    return function Box() {
      squirrel = payload;
      flag = true;
    }
  }

  function Unsealer(box) {
    flag = false; squirrel = null;
    box();
    caja.require(flag, ...);
    return squirrel;
  }

  return caja.freeze({
    seal: Sealer,
    unseal: Unsealer
  });
}

```

Figure 13: Rights Amplification. xxx

## 5.4 The Caja Library

TODO To be written

### 5.5 Examples

TODO To be written

## 6 Related work

### 6.1 Browser Shield

### 6.2 ADsafe

TODO To be written

## 7 Conclusions

TODO To be written

```

function Mint(name) {
  caja.requireType(name, 'string');
  var brand = Brand(name);
  return function Purse(balance) {
    caja.requireNat(balance);
    function decr(amount) {
      caja.requireNat(amount);
      balance =
        caja.requireNat(balance - amount);
    }
    return caja.freeze({
      getBalance: function() {
        return balance; },
      makePurse: function() {
        return Purse(0); },
      getDecr: function() {
        return brand.seal(decr); },
      deposit: function(amount, src) {
        var newBal =
          caja.requireNat(balance+amount);
        var box = src.getDecr();
        brand.unseal(box)(amount);
        balance += newBal;
      }
    });
  }
}

```

Figure 14: The MintMaker Example. xxx

## 8 Acknowledgements

We thank Dirk Balfanz, Bruno Bowden, Doug Crockford, Jed Donnelley, Brendan Eich.

## A Tables

### A.1 Property access

### A.2 Function calling

## References

- [1] Mashup Security Approaches. [openajax.org/member/wiki/Mashup\\_Security\\_Approaches](http://openajax.org/member/wiki/Mashup_Security_Approaches).
- [2] J. B. Dennis and E. C. V. Horn. Programming Semantics for Multiprogrammed Computations. Technical Report MIT/LCS/TR-23, M.I.T. Laboratory for Computer Science, 1965.
- [3] ECMA. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, Dec. 1999.
- [4] I. K. S. L. Garfinkel. Bitfrost: the One Laptop per Child Security Model. *Symposium On Usable Privacy and Security*, 2007.
- [5] A. H. Karp. Authorization-based access control for the services oriented architecture. *c5*, 0:160–167, 2006.
- [6] M. Košík. Backwater Operating System, 2007. [altair.dcs.elf.stuba.sk:60001/mediawiki/upload/2/2b/Backwater.pdf](http://altair.dcs.elf.stuba.sk:60001/mediawiki/upload/2/2b/Backwater.pdf).
- [7] A. M. Mettler and D. Wagner. The Joe-E Language Specification (draft). Technical Report UCB/EECS-2006-26, EECS Department, University of California, Berkeley, March 17 2006.
- [8] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

Caja expression	translates to ES3 code equivalent to
<code>with</code>	<code>/*rejected*/</code>
<code>this.id__</code>	<code>/*rejected in all positions*/</code>
<code>foo.id_</code>	<code>/*rejected in all positions*/</code>
<code>local__</code>	<code>/*rejected in all positions*/</code>
<code>glob_</code>	<code>/*rejected in all positions*/</code>
<code>this.id_</code>	<code>/*unaltered*/</code>
<code>this.id</code>	<code>____.readProp(this,"id")</code>
<code>foo.id</code>	<code>____.readPub(foo,"id")</code>
<code>this[bar]</code>	<code>____.readProp(this,bar)</code>
<code>foo[bar]</code>	<code>____.readPub(foo,bar)</code>
<code>glob</code>	<code>___OUTERS____.glob</code>
<code>bar in this</code>	<code>(bar in this) &amp;&amp; _____.canReadProp(this,bar)</code>
<code>bar in foo</code>	<code>(bar in foo) &amp;&amp; _____.canReadPub(foo,bar)</code>
<code>for (key in this) {...}</code>	<code>for (key in this) {if (canEnumProp(this,key)) {...}}</code>
<code>for (key in foo) {...}</code>	<code>for (key in foo) {if (canEnumPub(foo,key)) {...}}</code>
<code>this.id = baz</code>	<code>____.setProp(this,"id",baz)</code>
<code>foo.id = baz</code>	<code>____.setPub(foo,"id",baz)</code>
<code>this[bar] = baz</code>	<code>____.setProp(this,bar,baz)</code>
<code>foo[bar] = baz</code>	<code>____.setPub(foo,bar,baz)</code>
<code>glob = baz</code>	<code>___OUTERS____.glob = baz</code>
<code>var glob = baz</code>	<code>___OUTERS____.glob = baz</code>
<code>var glob</code>	<code>___OUTERS____.glob = undefined</code>
<code>delete this.id</code>	<code>____.deleteProp(this,"id")</code>
<code>delete foo.id</code>	<code>____.deletePub(foo,"id")</code>
<code>delete this[bar]</code>	<code>____.deleteProp(this,bar)</code>
<code>delete foo[bar]</code>	<code>____.deletePub(foo,bar)</code>
<code>delete glob</code>	<code>____.require(delete ___OUTERS____.glob,...)</code>

Figure 15: Translating Property Access. Under the assumption that the Caja runtime environment is as specified, the Caja translator generates translations equivalent to those specified above, but inlined and optimized where possible. The meaning of translating is thereby determined by the specification of these entry points into the Caja runtime library. Where we show a translation apparently duplicating an expression, the translator instead introduces temporary variables as needed so that each expression evaluates exactly as many times and in the same order as in the original.

Methods of ___	method body
require(test,complaint)	if (test) { return true; } throw new CajaRuntimeError(complaint);
canRead(obj,name)	return !!obj[name+"_canRead___"];
canEnum(obj,name)	return !!obj[name+"_canEnum___"];
canCall(obj,name)	return !!obj[name+"_canCall___"];
canSet(obj,name)	return !!obj[name+"_canSet___"];
canDelete(obj,name)	return !!obj[name+"_canDelete___"];
allowRead(obj,name)*	obj[name+"_canRead___"] = true;
allowEnum(obj,name)*	allowRead(obj,name); obj[name+"_canEnum___"] = true;
allowCall(obj,name)*	obj[name+"_canCall___"] = true;
allowSet(obj,name)*	require(!isFrozen(obj),...); obj[name+"_canCall___"] = false;
allowDelete(obj,name)*	allowEnum(obj,name); obj[name+"_canSet___"] = true; require(!isFrozen(obj),...); obj[name+"_canDelete___"] = true; /*other bookkeeping yet to be determined*/
hasOwnProp(obj,name)	return Object.prototype.hasOwnProperty.call(obj,name);
isJSONContainer(obj)	var Constr = directConstructor(obj); return Constr === Object    Constr === Array;
isFrozen(obj)	return hasOwnProp(obj,"___FROZEN___");
freeze(obj)*	for (k in obj) { if (endsWith(k,"_canSet___")    endsWith(k,"_canDelete___")) { obj[k] = false; } } obj.___FROZEN___ = true; return obj;

Figure 16: Hidden Attributes. These methods handle the concrete representations of object and property attributes. Only the methods marked with a \* should called by Javascript code during initialization of the embedding app, in order to express taming decisions. All objects that are reachable from the ES3 shared scope should be frozen, so that this shared scope becomes transitively read-only to all Caja code.

Methods of ___	method body
canReadProp(self,name)	if (endsWith(name,"__")) { return false; } return canRead(self,name);
readProp(self,name)	return canReadProp(self,name) ? self[name] : undefined;
canReadPub(obj,name)	if (endsWith(name,"_")) { return false; } if (canRead(obj,name)) { return true; } if (!isJSONContainer(obj)) { return false; } if (!hasOwnProp(obj,name)) { return false; } allowRead(obj,name); return true; /*memoize*/
readPub(obj,name)	return canReadPub(obj,name) ? obj[name] : undefined;
canEnumProp(self,name)	if (endsWith(name,"__")) { return false; } return canEnum(self,name);
canEnumPub(obj,name)	if (endsWith(name,"__")) { return false; } if (canEnum(obj,name)) { return true; } if (!isJSONContainer(obj)) { return false; } if (!hasOwnProp(obj,name)) { return false; } allowEnum(obj,name); return true; /*memoize*/
canSetProp(self,name)	if (endsWith(name,"__")) { return false; } if (canSet(self,name)) { return true; } return !isFrozen(self);
setProp(self,name,val)	require(canSetProp(self,name),...); allowSet(self,name); /*grant*/ return self[name] = val;
canSetPub(obj,name)	if (endsWith(name,"_")) { return false; } if (canSet(obj,name)) { return true; } return !isFrozen(obj) && isJSONContainer(obj);
setPub(obj,name,val)	require(canSetPub(obj,name),...); allowSet(obj,name); /*grant*/ return obj[name] = val;
deleteProp(self,name)	require(canDeleteProp(self,name),...); /*XXX Bookkeeping yet to be determined*/ return require(delete self[name],...);
deletePub(obj,name)	require(canDeletePub(obj,name),...); require(isJSONContainer(obj),...); /*XXX Bookkeeping yet to be determined*/ return require(delete obj[name],...);

Figure 17: Property Access. The calls to `allowRead` and `allowEnum` merely memoize a query result. The calls to `allowSet` track the implications of side effects.

Global ES3 non-constructor	Property	Taming
<code>NaN</code>		ok
<code>Infinity</code>		ok
<code>undefined</code>		ok
<code>eval</code>		hidden
<code>parseInt</code>		ok
<code>parseFloat</code>		ok
<code>isNaN</code>		ok
<code>isFinite</code>		ok
<code>decodeURI</code>		ok
<code>decodeURIComponent</code>		ok
<code>encodeURI</code>		ok
<code>encodeURIComponent</code>		ok
<code>Math</code>		ok
	<code>random</code>	ok*
	all others in ES3	ok

Figure 18: Taming ES3 Global Non-Constructors. Except for `eval`, all non-constructors specified by ES3 are visible in Caja’s outer scope as immutable objects. Note that `Math.random` is not actually immutable, and therefore neither is `Math` nor Caja’s outer scope itself. We allow it anyway for reasons explained in the text.

- [9] J. A. Rees. A Security Kernel Based on the Lambda-Calculus. Technical report, Massachusetts Institute of Technology, 1996.
- [10] M. Seaborn. Plash: The Principle of Least Authority Shell, 2005. [plash.beasts.org/](http://plash.beasts.org/).
- [11] M. Stiegler. Emily, a High Performance Language for Secure Cooperation, 2006. [skyhunter.com/marcs/emily.pdf](http://skyhunter.com/marcs/emily.pdf).
- [12] M. Stiegler, A. H. Karp, K.-P. Yee, and M. S. Miller. Polaris: Virus Safe Computing for Windows XP. Technical Report HPL-2004-221, Hewlett Packard Laboratories, 2004.
- [13] D. Wagner and E. D. Tribble. A Security Analysis of the Combex DarpaBrowser Architecture, Mar. 2002. [combex.com/papers/darpa-review/](http://combex.com/papers/darpa-review/).
- [14] H. J. Wang, X. Fan, C. Jackson, and J. Howell. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP’07)*. ACM, Oct. 2007.

Caja expression	translates to ES3 code equivalent to
<code>/*caja module body*/</code>	<code>___module(function(___OUTERS___) {     /*translated module body*/ });</code>
<code>new F(a...)</code>	<code>___callNew(F, [a...])</code>
<code>this.id(a...)</code>	<code>/*unaltered*/</code>
<code>this.id(a...)</code>	<code>___invokeProp(this, "id", [a...])</code>
<code>foo.id(a...)</code>	<code>___invokePub(foo, "id", [a...])</code>
<code>foo(a...)</code>	<code>foo.call___(___USELESS, a...)</code>
<code>function F(a...) {     F.Super.call(this, b...);     ...this...}</code>	<code>var F = ___ctor(function(a...) {     ___enterDerived(F, this);     F.Super.call___(this, b...);     ...this...})</code>
<code>function F(a...) {     ...this...}</code>	<code>var F = ___ctor(function(a...) {     ___enterBase(F, this);     ...this...})</code>
<code>function(a...) {     ...this...}</code>	<code>___method(F, function(a...) {     ___enterMethod(arguments.callee, this);     ...this...})</code>
<code>function foo(a...) {     ...}</code>	<code>var foo = ___freeze(function(a...) {     ...; return undefined;})</code>
<code>function(a...) {     ...}</code>	<code>___freeze(function(a...) {     ...; return undefined;})</code>
<code>arguments.callee ...arguments...</code>	<code>/*unaltered, to allow anonymous recursion*/ var ___args___ = ___args(arguments); ...___args_____</code>
<code>/pattern/</code>	<code>new RegExp("pattern")</code>

Figure 19: Translating Callers and Callees. A translated Caja module can be loaded/evaluated once, creating an anonymous plugin-maker function. Each time the plugin-maker representing a module is called, it makes a new plugin. Unless their creator puts them in contact, even plugins made by the same plugin-maker are isolated from each other. A function that falls off the end is rewritten to return `undefined` explicitly, in case it is called as a constructor.

Methods of ___	method body
isCtor(Constr)	return !!Constr.___CONSTRUCTOR___;
isMethod(meth)	return !!meth.___METHOD_OF___;
isMethodOf(meth,receiver)	if (isMethod(meth)) { return receiver instanceof meth.___METHOD_OF___; } else { return !isCtor(meth); }
ctor(Constr)*	require(typeof Constr === "function",...); Constr.___CONSTRUCTOR___ = true; return Constr; /*translator freezes Constr later*/
method(Constr,meth)*	require(typeof meth === "function",...); require(isCtor(Constr),...); meth.___METHOD_OF___ = Constr; return freeze(meth);
wrapMethod(Constr,name,meth)*	require(name in Constr.prototype,...); meth.___ORIGINAL___ = Constr.prototype[name]; Constr.prototype[name] = meth; return method(Constr,meth);
makeRaw(Constr)	function F() { this.___RAW___ = true; } F.prototype = Constr.prototype; return new F();
cookIfRaw(obj)	return delete obj.___RAW___;

Figure 20: Other Hidden Annotations. xx.

Methods of ___	method body
callNew(Constr,args)	require(!isMethod(Constr),...); var result = Constr.apply(makeRaw(Constr), args); cookIfRaw(result); /*remove RAW flag as soon as possible*/ return result;
enterBase(Constr,self)	require(self instanceof Constr,...); require(cookIfRaw(self),...);
enterDerived(Constr,self)	require(self instanceof Constr,...); require(isRaw(self),...);
enterMethod(meth,self)	require(isMethodOf(meth,self),...); require(!cookIfRaw(self),...);
args(original)	return freeze(Array.prototype.slice.call___(original,0));

Figure 21: Context Switching. Javascript code must not use Javascript's `new` on Caja code. Beyond this, no matter whether caller or callee is in Javascript or Caja, a constructor can be called as a constructor using `new`, a method can be called as a method, and a function can be called as a function, method, or constructor.

Global ES3 constructor	Property	Taming
constructor		default ok
constructor.prototype		default ok
	constructor	hidden
	toString	default ok
	toLocaleString	default ok
	valueOf	default ok
instances	length	default ok
Object.prototype	hasOwnProperty	wrapped
	isPrototypeOf	ok
	propertyIsEnumerable	wrapped
Function		hidden
Function.prototype		ok?
	apply	wrapped
	call	wrapped
instances	prototype	ok
	length	ok
Array.prototype	concat	ok
	join	ok
	pop	wrapped
	push	wrapped
	reverse	wrapped
	shift	wrapped
	slice	ok
	sort	wrapped
	splice	wrapped
	unshift	wrapped
	indexOf	added
	lastIndexOf	added
String	fromCharCode	ok
String.prototype	match	wrapped
	replace	wrapped
	search	wrapped
	split	wrapped
	all others in ES3	ok

Figure 22: Taming ES3 Global Constructors, Part 1. The first section above shows the taming decisions that apply by default to global ES3 constructors, their prototypes, and their instances, unless stated otherwise in a specific table entry. The `Array` methods are wrapped to ensure they don't violate Caja's mutability constraints.

Global ES3 constructor	Property	Taming
<b>Boolean</b>		ok
<b>Number</b>	MAX_VALUE	ok
	MIN_VALUE	ok
	NaN	ok
	NEGATIVE_INFINITY	ok
	POSITIVE_INFINITY	ok
<b>Number.prototype</b>	toFixed	ok
	toExponential	ok
	toPrecision	ok
<b>Date</b>		ok*
	parse	ok
	UTC	ok
<b>Date.prototype</b>	to*String all in ES3	ok
	toISOString	added
	get* all in ES3	ok
	set* all in ES3	wrapped
<b>RegExp.prototype</b>	exec	wrapped
	test	wrapped
<b>instances</b>	source	ok
	global	ok
	ignoreCase	ok
	multiline	ok
	lastIndex	ok
<b>Error.prototype</b>	name	ok
	message	ok
<b>*Error</b>	all in ES3	ok
<b>*Error.prototype</b>	all in ES3	ok

Figure 23: Taming ES3 Global Constructors, Part 2. The **Date** constructor itself gives ambient read-only access to the current time, and is therefore not immutable. We allow it anyway for reasons explained in the text.