

Caja

Safe active content in sanitized JavaScript

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

January 15, 2008

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 re-

quests using its public interface. But how an object reacts to a request 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*.¹ 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 affect 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 world outside the browser 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. Caja² 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.

¹ Thanks to Mark Lillibridge for this formulation.

² *Caja*, pronounced “KA-hah”, is Spanish for “box”. With Caja, capabilities attenuate JavaScript authority.

Forbidden names. Caja rejects all names ending with “_” (double underscore). This gives the 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 environment are *immutable*, or transitively frozen.

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

Internal names. Property names ending in “_” (single underscore) serve as protected instance variables. Such names can only appear to the right 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 “this” 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”.

Caja supports stereotyped class-like usage of constructors and methods, but prohibits certain other dangerous usage patterns. A constructor can only be called *as a constructor* using `new`, or by a directly derived constructor to initialize a derived instance. An object’s methods can only be called as methods of *that* object, even

```
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 are not restricted.

when calling the method reflectively using `call`, `apply`, or `bind`.

Sharp knives removed. Caja contains no “with” or “eval”. Caja includes a safe JSON library to support the most common use of `eval`, and a safe `caja.eval` for evaluating code in the *Cajita* subset of Caja. *Cajita* is essentially the subset of Caja without “this”.

Hopefully, this is all the casual Caja programmer needs to know to get started. Section 2 is a partisan history of access control on the web, 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.

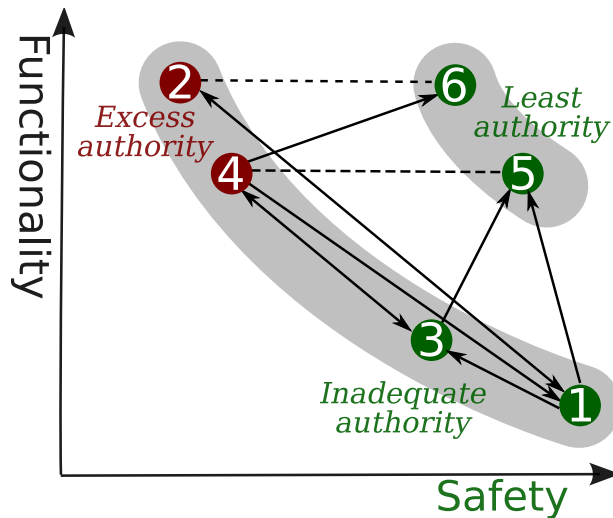


Figure 2: The Evolving Authority of Active Content. Identity-centric access controls have led to thrashing between lost functionality and lost safety. To have both, we need to provide *least authority*: adequate authority for desired functionality without excess authority which invites abuse.

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 attempts 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-centric access control [4], 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

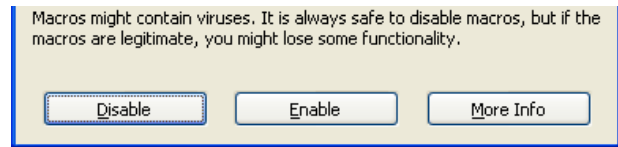


Figure 3: Only Bad Choices. When documents contain scripts, users can disable themselves from getting any work done ① or enable scripts to destroy all their other work ②.

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.)

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-centric 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 distinguish 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-centric 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

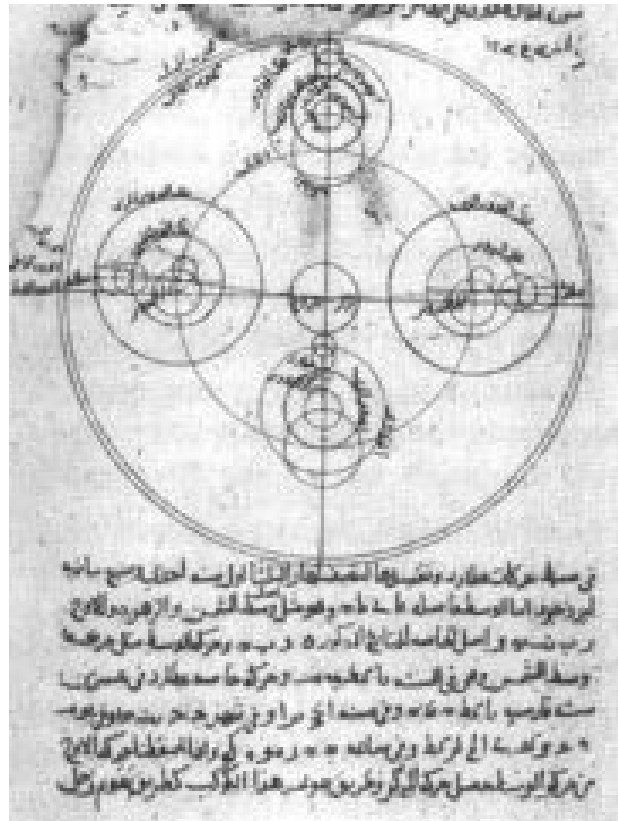


Figure 4: Ptolemy’s epicycles. Ptolemy attempted to model the motion of the heavenly bodies using only circles. With each discovery that the model didn’t fit, yet another layer of circle was added to adjust. By contrast, Kepler’s ellipses fit the problem directly, with no need for endless additional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

an authorization-centric model such as object-capabilities [1]. The object-capability alternative naturally supports POLA, the principle of least authority, shown in the upper right of Figure 2. 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 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, Perl and others demonstrates that existing memory safe languages often already contain an expressive object-capability subset [7, 9, 11, 5, 6, respectively]. We refer to the object-capability subset of JavaScript as *Caja*. This subset 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 sanitizer 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, 3]. A web app, on detecting the presence of a powerbox, could offer to edit a local file chosen by the user ④→⑥.

3 Subsetting JavaScript

Caja defines a subset of JavaScript both syntactically and semantically. The Caja sanitizer 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 sanitizer translates the JavaScript it accepts into *cajoled*³ *Javascript*—Javascript with additional runtime checks. To facilitate development, it is easy to write a program in the Caja subset of JavaScript so it can run correctly whether it is cajoled or not.

Our starting point is JavaScript as documented in the third edition of the EcmaScript 262 standard [2]; hereafter *ES3*⁴. The remainder of this document explains the differences between Caja—the JavaScript subset accepted by the Caja sanitizer—and ES3. Other documents will explain the interface between cajoled and uncajoled JavaScript, and Caja’s sanitization of the remaining elements of active web con-

³ We thank Pat Patterson for this term.

⁴ ES3 is approximately a bit more than JavaScript 1.4 and a bit less than JavaScript 1.5.

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

Figure 5: A Cajita Counter. Each call to `Counter()` 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. A counter object as a whole, as well as each of its pseudo-methods, are thus proper protected capabilities. Someone with access only to a counter’s `incr` function can increment *that* counter and observe the result, but not do anything else.

tent: 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 sanitizer as *Caja web content*, and to the sanitizer’s corresponding output as *Cajoled web content*.

3.1 The OS analogy

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 cajoled JavaScript output by the Caja sanitizer. All un-

trusted scripts must be provided as Caja source code, to be statically verified and cajoled by the Caja sanitizer. The sanitizer’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 environment 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 Caja sanitizer 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 cajoled together defines a *Caja module*. All code within the same module share a global environment, but distinct modules see disjoint global environments. The Caja sanitizer 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.

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

```

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

var ptA = Point(3, 5);
var ptB = Point(4, 7);

```

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.

Implicit mutable state. Some base JavaScript objects, such as `Array.prototype`, are implicitly reachable even without naming any global variable names. Even after global environment problems are fixed, the mutability of these objects would prevent isolation.

Lack of encapsulation. To support the “context switching” criterion explained in section 3.1, 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 environment.

Foreign for/in loops. JavaScript’s `for/in` loop enumerates the names of all an object’s prop-

```

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

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 `that` as its first parameter. The method-like functions can use `that` 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 “subclasses” such as `WobblyPointMixin` in Figure 8. If the class is non-abstract, it should also have a pseudo-constructor function such as `Point` for making direct instances.

erties, 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

```

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

```

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

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 make 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 or 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 9, Safari 3, and their successors. Fortunately, these

browsers do conform closely to ES3. Later versions of Caja may specify larger subsets of ES3.

Multiple worlds. As with many languages, each instantiation of a JavaScript language world creates a set of primordial objects (like `Object.prototype`) that are global to that world. Unlike other languages, JavaScript is built to support multiple interacting worlds. For example, in the browser environment, a new JavaScript world is created for each `iframe`. An object from one `iframe` can hold a direct reference to an object from another `iframe` of the same origin. This leads to some surprises. Even if `x` holds an array, `x instanceof Array` may evaluate to false because `x` is an instance of the `Array` from a different JavaScript world.

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. In JavaScript, reading a non-existent property returns `undefined` rather than throwing an exception. The JavaScript *feature testing* pattern relies on this behavior. Since, in this case, the program naturally notices the problem anyway, Caja does not turn this case into a thrown exception.

The above point about “Silent errors” is another reason to avoid the monolithic kernel approach. Web apps in uncajoled JavaScript are vulnerable to any malicious active content that finds a way to provoke a silent error and exploit the resulting inconsistency.

3.3 A fail-stop subset

Because Caja failures aren't silent, Caja is a *fail-stop subset*⁵ of ES3: While a Caja program has not explicitly indicated a failure, it executes within ES3's semantics. By *indicate a failure*, we mean either throwing an exception or returning `undefined` for a property read. The *Gotchas* sections 4.5 and 5.4 detail minor ways in which Cajita and Caja are not quite fail-stop subsets of ES3. A *Caja-compliant* JavaScript program is one which

1. is statically accepted by the Caja sanitizer,
2. does not provoke Caja-induced failures when run cajoled, and
3. avoid these gotchas.

Such a program should have the same semantics whether run cajoled or not.

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".⁶

The Caja runtime will provide a safe `caja.eval` operation. For `caja.eval` to accept Caja code, the Caja sanitizer would need to be written in JavaScript and included in the Caja download. To minimize download size, `caja.eval` will instead accept only Cajita code.

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

Record An object whose prototype's "constructor" property is `Object`, i.e., under normal conditions, an object inheriting

directly from `Object.prototype`. Records are normally created using the `{...}` syntax.

Array An object whose prototype's "constructor" property is `Array`, i.e., under normal conditions, an object inheriting directly from `Array.prototype`. Arrays are normally created using the `[...]` syntax.

JSON Container. A record or array. These are the non-primitive objects that can be directly expressed in JSON syntax.

Invocation A function can be invoked

- as a function (`foo(a...)`),
- as a method (`foo.m(a...)`),
- as a constructor (`new Foo(a...)`), or
- reflectively (by calling its `call`, `apply`, or `bind` methods).

Simple-functions. A function whose body does not mention `this` is a *simple-function*. A simple-function can be either named or anonymous. 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, Caja functions are implicitly frozen. The Caja runtime library additionally provides an explicit operation for freezing JSON containers: `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`,

⁵ We thank Dan Rabin for this formulation.

⁶ The design of Cajita was inspired by Doug Crockford's *ADsafe*.

$$\begin{aligned}
F(\dots) &\equiv \text{new } F(\dots) & (1) \\
&\equiv F.\text{call}(v, \dots) & (2) \\
&\equiv F.\text{apply}(v, [\dots]) & (3) \\
&\equiv F.\text{bind}(v)(\dots) & (4) \\
F(\dots_1, \dots_2) &\equiv F.\text{bind}(v, \dots_1)(\dots_2) & (5) \\
x.m &\equiv \text{true} \ \&\& \ x.m & (6) \\
(x.m)(\dots) &\equiv (\text{true} \ \&\& \ x.m)(\dots) & (7) \\
\{\dots\} &\equiv (\text{function}()\{\text{return } \dots\})() & (8)
\end{aligned}$$

Figure 9: Cajita Regularities. Given that F is a simple-function, $x.m$ holds a simple-function, and v is an expression with no effects and stable value (such as a variable reference), then most of these equivalences hold in Caja as well as Cajita. Equation (8) holds in general only in Cajita. See section 4.1 for further qualifying conditions.

all objects that are globally or implicitly accessible to all Caja programs are immutable. We discuss these exceptions in section 4.4.

4.1 Cajita regularities

The regularities in Figure 9 apply when calling simple-functions, whether the calling code is in Cajita or Caja. When calling other functions, only the weaker *Caja regularities* shown in Figure 12 apply. The regularities in both sections are often stronger than ES3, but are all within a fail-stop subset of ES3.

- Equation (1) of Figure 9 states that the `new` keyword does not change the meaning of calling a simple-function. This holds only for simple-functions that explicitly `return` a value. As in uncajoled JavaScript, if a simple-function instead implicitly returns, it will return `undefined` when called without `new`, but will return a useless object when called with `new`.
- When calling the `call`, `apply`, or `bind` method of a simple-function, the first argument is ignored.

- The `apply` method differs from `call` only in packaging all arguments together into a list.
- A single-argument `bind` of a simple-function returns a function with equivalent invocation behavior—a function that behaves the same, whether called as a function, as a constructor, as a method, or reflectively.
- When `bind` has additional arguments, it returns a new function representing F carried over these additional arguments.
- In JavaScript, when the left operand of an `&&` expression evaluates to `true`, the `&&` expression as a whole evaluates to the value of its right operand. Therefore, you might expect Equation (6) to hold in general. The next item sheds light on why it does not hold when the value of the right operand is a non-simple-function.
- When the value of a property is a function, the expression on the left of Equation (7) will call it as a method on x , whereas the expression on the right will first extract the property value and then call it as a function. Fortunately, when this value is a simple-function, these two forms of invocation have the same meaning.
- In Cajita only, a simple-function correctly abstracts over the code it contains, given that this code makes no reference to `arguments`.

4.2 Cajita static restrictions

Any source code statically accepted by the Caja sanitizer is a *legal Caja program*. A legal Caja program satisfying additional static restrictions is also a *legal Cajita program* and will be accepted by the Cajita sanitizer. A Caja-compliant JavaScript program that is also a legal Cajita program is a *Cajita-compliant JavaScript program*—it will have the same semantics whether uncajoled, cajoled by the Caja sanitizer, or cajoled by the Cajita sanitizer.

The static restrictions immediately below apply to both Caja and Cajita. This is followed by the additional static restrictions specific to Cajita.

```

function Brand() {
  var flag = false;
  var squirrel = null;

  return caja.freeze({
    seal: function(payload) {
      function box() {
        squirrel = payload;
        flag = true;
      }
      box.toString = function() {
        return "(box)";
      };
      return box;
    },
    unseal: function(box) {
      flag = false; squirrel = null;
      box();
      if (!flag) { throw ...; }
      return squirrel;
    }
  });
}

```

Figure 10: Rights Amplification. Each brand has a `seal` and `unseal` function, acting like a matched encryption and decryption key. Sealing an object returns a sealed `box` that can only be unsealed by the corresponding `unseal` function. The implementation technique shown here is due to Marc Stiegler.

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 cajole to conforming ES3 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 cajoled 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 sanitizer’s cajoled output and by the Caja runtime. Firefox reserves the double underscore for itself.

“new” is ok. Since Cajita does not have `this`, constructors, nor prototypes, `new` isn’t needed purely within Cajita. But since Cajita code must interoperate smoothly with Caja and uncajoled JavaScript code, `new` is considered a valid part of Cajita.

The following features are present in Caja in order to accommodate old code, rather than to enhance expressiveness. Since Cajita is for new code, in order to minimize the download size of the Cajita sanitizer, as well as to simplify the semantics of Cajita considered on its own, these features are absent from Cajita. Code containing these features is not legal Cajita.

“this”. The central difference between Caja and Cajita is that only Caja includes “`this`”.

Internal names. In Caja, an *internal name* is a property name ending in “`_`” (a single underbar). Such names are used for encapsulation in Caja but are prohibited in Cajita. Cajita’s only encapsulation mechanism is lexical scoping.

```

function Mint() {
  var brand = Brand();
  return function Purse(balance) {
    caja.enforceNat(balance);
    function decr(amount) {
      caja.enforceNat(amount);
      balance =
        caja.enforceNat(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.enforceNat(balance+amount);
        var box = src.getDecr();
        brand.unseal(box)(amount);
        balance += newBal;
      }
    });
  }
}

```

Figure 11: The MintMaker Example. Calling `Mint()` creates a `Purse` function for making purses holding new transferable units of a distinct “currency”. Given two purses of the same currency, one can transfer money between them, but one can’t violate conservation of currency.

Prototypes. In Caja and Cajita, if “`Foo`” is a function name, then static properties of the function can be initialized until the first time the function is used. Cajita prohibits access to `Foo`’s “**prototype**” property, and so prevents use of JavaScript’s prototype inheritance within Cajita.

“instanceof”. Without “`this`” and prototypes, Cajita has no need for `instanceof`. Rather,

JavaScript’s `typeof` is almost an adequate type discriminator for Cajita. But Cajita still needs a way to distinguish records from arrays. We could allow the conventional `x instanceof Array` expression, but it does not work correctly when `x` is an array from another cajoled JavaScript world, such as another `iframe` with the same origin. Instead we provide `caja.isArray(x)` as a correct alternative.

Literal RegExp syntax. In JavaScript implementations, the literal pattern syntax is often optimized into a static object with mutable state, violating isolation. The Caja sanitizer cajoles the `/pattern/` syntax to `new RegExp("pattern")`. In Cajita, the second form must be written explicitly.

for/in loops. Because of the confusing semantics of JavaScript’s `for/in` loops, these are absent from Cajita. Instead, Cajita code should enumerate the properties of `obj` by doing

```
caja.forEach(obj,function(v,k){...});
```

This code will reliably give the same results whether run cajoled or not. It will enumerate only the non-inherited publicly Caja-visible property value / property name associations of `obj`. If `obj instanceof Array`, then `k` will enumerate successive indexes into the array.

Semicolon insertion. Code which would provoke semicolon insertion is not legal Cajita.

Block-breaking scopes. Cajita variable names are visible only according to the intersection of ES3’s scoping rules and conventional Java-like block-level lexical scoping. This is essentially lexical scoping from the point of introduction with the restriction that a function cannot contain two definitions of the same variable name, even in two separate blocks. If JavaScript scope analysis and conventional block-level lexical scope analysis would disagree on the variable bindings of a given piece of code, then that code is not legal Cajita.

Coercing equality. JavaScript’s coercing rules for the “==” and “!=” operators are complex, accident prone, and not even transitive. Cajita only includes the equality operators “===” and “!==”.

4.3 Cajita dynamic restrictions

The following restrictions apply to both Caja and Cajita.

Frozen Functions. An anonymous simple-function is implicitly frozen. A named simple-function may be initialized, but is implicitly frozen immediately before its first non-initializing use or escaping occurrence. For example, the assignment to `box.toString` in Figure 10 will succeed, because it occurs before `box` is implicitly frozen by the following `return` statement. Initializing assignments can thus be considered declarative initializations rather than mutations.

Claim: No Caja program can cause a Caja-observable mutation of a function or of any object Caja considers frozen.

4.4 Outer environment restrictions

The following restrictions apply to both Caja and Cajita.

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

Caja’s *shared outer environment* contains the subset of the standard ES3 global environment 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 environment are transitively immutable, so that they provide no ambient authority to objects defined within this environment. The shared outer environment itself is frozen.

Each instantiation of a Caja module is a separate plugin. Each plugin executes with a new *per-plugin outer environment* which is mutable and in-

herits from Caja’s shared outer environment. Therefore, all objects within the same plugin can implicitly communicate and interfere with each other.

Claim: Two separate plugins, even if they instantiate the same module, are isolated from each other.

eval The Caja global environment has no “eval” function. Instead, “`caja.eval`” will evaluate Cajita source code (text or AST) with an explicitly provided set of variable bindings to serve as its initial outer environment, as `TODO(erights)` will be explained somewhere.

Function The JavaScript `Function` constructor is absent from the default Caja outer environment, and must not be available in Caja.

Restricted reflection The JavaScript `constructor` property of prototypical objects and functions is absent from Caja. The `prototype` property of functions can only be used in the limited ways shown in Figure 19. And the `call`, `apply`, and `bind` methods of functions cannot be replaced or overridden.

Claim: The restrictions stated in this document together make the `Function` object unreachable from Caja programs.

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 environment 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 Cajita gotchas

Caja seeks to define a fail-stop subset of ES3, as explained in section 3.3. However, it falls short of this goal in several minor ways. To write a correct program that executes correctly whether run cajoled or uncajoled, it should avoid these gotchas. In this section, we enumerate those gotchas relevant to the Cajita subset of Caja.

Snapshot “arguments”. In ES3, if `x` is the i 'th parameter of a function, assignments to `x` are visible as changes to `arguments[i]` and vice versa. In Caja, if “arguments” is mentioned, it is bound to a proper array snapshot of the arguments list when the function was entered, *not* an array-like object. In order for Caja to be a fail-stop subset of ES3, a future version of the Caja sanitizer will statically disallow assignments to any parameter variable within a function that mentions “arguments”. But in the initial Caja implementation, this minor gotcha remains.

Absent ReferenceError. In ES3, when a reference to an undefined variable is evaluated as an expression, a `ReferenceError` is thrown. The Caja sanitizer cajoles a reference to an undefined variable into a reference to an outer variable. Given the current cajoling rules, a reference to an undefined outer variable will evaluate to `undefined` rather than throwing a `ReferenceError`. We expect to repair this issue in a future Caja spec and implementation.

5 Caja Specification

Whereas Cajita is a small subset of JavaScript meant to support new code, Caja is a large subset of JavaScript meant to ease the porting of old JavaScript code and practices. Cajita is small enough that its security properties can be understood. Caja seeks to accept as large a subset of JavaScript as is practical without losing the security properties provided by Cajita. In this section, we explain only the remaining elements of Caja beyond the elements of Cajita already explained.

To explain the remaining elements of Caja, 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 function other than `Array` or `Object`.

Prototypical objects. As in ES3, a constructed object's implicit *prototype*—the object it directly inherits from—is the value of the “`.prototype`” property of the function which constructed it (which must have been called with “new”). In Caja, these *prototypical objects* are not first-class. When a function is implicitly frozen, so is its `.prototype`. Until then, both it and its `.prototype` may be initialized.

Constructors. A named function whose body mentions “this” is a *constructor*.

Methods. An anonymous function whose body mentions “this” is a *method*.

5.1 Caja regularities

The regularities shown in Figure 12 apply when Caja code calls any Caja function *other than* a constructor. These regularities are often stronger than ES3, but are all within a fail-stop subset of ES3.

- The code on the left of Equation (9) of Figure 12 calls `x.m` as a method on `x`. The code on the right first extracts the value of `x.m`. When `x.m` is a method, the extracted value is an *attached*

$\begin{aligned} x.m(\dots) &\equiv (\text{true}\&\&x.m).call(x,\dots) &(9) \\ &\equiv x.m.call(x,\dots) &(10) \\ &\equiv x.m.apply(x,[\dots]) &(11) \\ &\equiv x.m.bind(x)(\dots) &(12) \\ x.m.bind(x) &\equiv (\text{true} \&\& x.m).bind(x) &(13) \\ x.m(\dots_1,\dots_2) &\equiv x.m.bind(x,\dots_1)(\dots_2) &(14) \\ \{\dots\} &\equiv (\text{function}()\{ &(15) \\ &\quad \text{return} \dots\}).call(\text{this}) \end{aligned}$	<pre>function Point(x, y) { this.x_ = x; this.y_ = y; } Point.prototype.toString = function() { return "<" + this.getX() + "," + this.getY() + ">"; }; Point.prototype.getX = function() { return this.x_; }; Point.prototype.getY = function() { return this.y_; }; var ptC = new Point(3, 5); var ptD = new Point(4, 7);</pre>
---	---

Figure 12: Caja Regularities. In Caja, given that `x.m` is associated with either a simple-function or a method, then these equivalences hold. Equation (15) holds only in the absence of variable name conflicts. See section 5.1 for further qualifying conditions.

method whose attachment is `x`. When `x` is an expression with no effects and a stable value (such as a variable reference), the code on the right then calls the attached method’s `call` method with its attachment and the original arguments. These two calls are equivalent.

- The `apply` method differs from `call` only in packaging all arguments together into a list.
- Binding an attached method to its attachment yields a conventional bound method—a simple-function of the remaining arguments which calls the original method as a method on its attachment.
- When `bind` has additional arguments, it returns a new function representing `F` curried over these additional arguments.
- An inner attached method, as shown in Figure 16, can only appear within an enclosing method or constructor. Equation (15) of Figure 12 states that, in Caja, an inner attached method correctly abstracts over the code it contains. This holds when this code makes no reference to `arguments` and does not define any vari-

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

var ptC = new Point(3, 5);
var ptD = new Point(4, 7);
```

Figure 13: A Caja Point. The point example, written in this common class-like pattern of Javascript programming, is valid Caja. `Point` is frozen by its first use, after which neither `Point` nor `Point.prototype` can be further initialized.

able names that conflict with variables defined or used by its enclosing method or constructor.

5.2 Caja static restrictions

Any source code statically accepted by the Caja sanitizer is a *legal Caja program*. The following syntactic explains why a program may instead be statically rejected.

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

Constructor names. A Caja constructor can only be called as a constructor using `new`, in order to instantiate a direct instance, or reflectively using `call` only by derived constructors, in order to begin initializing a derived instance. A

```

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

```

Figure 14: A Brief Caja Point. Caja also accepts this more compact pattern for initializing a top-level prototype all at once.

derived constructor must begin by calling its super constructor’s `call` method with `this` as the first argument and no `this` appearing elsewhere in the argument list. This use of `call` is exempt from the normal translation rules.

Like a named simple-function, a constructor and its `“prototype”` may be initialized, but are implicitly frozen on first use. A constructor must not contain an explicit `return`.

Methods. To avoid the confusions regarding `“this”`, Caja methods may only appear in the positions marked `“member”` in Figure 19, as shown by example in Figures 13, 14, 15, and 16. Methods may thus be used to initialize properties of prototypes, or to abstract part of the logic of an enclosing method or constructor.

Although constructors are normally frozen and the `“prototype”` property of functions is generally not accessible, we allow the patterns shown in Figures 13 and 14 for declaring a constructor, initializing it, and initializing its prototype.

If the first argument to `“Caja.def”` is a function name, this is considered an initializing use, and so does not implicitly freeze that function.

5.3 Caja dynamic restrictions

The following additional dynamic restrictions are relevant are relevant to Caja code.

Frozen prototypes. In Caja, until a function is frozen, both it and the value of its `“prototype”` property may be initialized. When a function is frozen, so is the value of its `“prototype”` property. Therefore, only the instances at the leaves of the JavaScript inheritance tree may remain unfrozen. Initializing assignments to a function’s `“prototype”` can thus be considered declarative initializations rather than mutations.

Claim: No Caja program can cause a Caja-observable mutation of a prototypical object.

Well formed inheritance. JavaScript provides an interesting set of primitives for building non-standard inheritance arrangements. Many of these arrangements will break assumptions in other code. In practice, these primitives are used in a particular arrangement in which, for example, for all functions `F`, `F.prototype.constructor === F`. Caja allows only this classical inheritance pattern, so that Caja code and the Caja implementation can rely on it.

Shape change. When one adds or deletes properties of an object, we can describe this as changing the shape of the object. Of course, no one can change the shape of a frozen object. Anyone with access to a non-frozen JSON container may freely change its shape. A constructed object can directly change its own shape, by assignment or `delete` using `this`. Clients of a constructed object cannot directly change its shape. But since a constructed object can directly change its own shape, it can provide methods enabling its clients to ask it to change its shape. In other words, a constructed object has control of its own shape. Adding a property that overrides an inherited property is considered a shape change, so only a constructed object may do this directly for itself. If a constructed object does create a public own property, its clients can directly assign to it.

Non-reflective constructors Any attempt to call a constructor's `call`, `apply`, or `bind` methods must fail, except for the statically exempted use of `call` mandated in section 5.2 for derived constructors.

Attached methods Any attempt to obtain a method as a value will instead yield an *attached method*. If `x.foo(...)` would directly call a method, then `x.foo` will return that method as attached to `x`. As shown in Figure 16, a method appearing within an enclosing constructor or method will evaluate to an inner attached method—a method attached to the value of `this` in its enclosing lexical context. An attached method can only be invoked by calls that bind its `this` to its attachment, whether called as a method on its attachment, or called reflectively by providing its attachment as the first argument of `call`, `apply`, or `bind`. Since calling an attached method either fails or acts like calling the original method, an attached method behaves within a fail-stop subset of the behavior associated with the original method.

5.4 Caja gotchas

Caja seeks to define a fail-stop subset of ES3, as explained in section 3.3. However, it falls short of this goal in several minor ways. To write a correct program that executes correctly whether run cajoled or uncajoled, it should avoid these gotchas. In this section, we enumerate those remaining gotchas relevant specifically to Caja.

Bare for/in loops. More properties are visible and enumerable to uncajoled programs than cajoled programs. To write a program which will see the same properties whether run cajoled or not, write the following instead:

```
for (var k in obj) {
  if (caja.canEnumPub(obj,k)) {
    ...k...obj[k]...
  }
}
```

```
function WobblyPoint(x, y) {
  Point.call(this, x, y);
}
Caja.def(WobblyPoint, Point, {
  getX: function() {
    return Math.random() +
      Point.prototype.getX.call(this);
  }
});
```

Figure 15: A Caja Subclass. `caja.def` supports classical inheritance. The second argument serves as a “superclass”. The third argument provides instance members including methods. A fourth optional argument (unshown) provides static members. When the superclass argument is a function name, it can be used as shown for *super* construction and method calls. These occurrences of `call` are exempt from the normal cajoling rules.

This conditional does not affect the behavior of cajoled programs, so programs that only need to run cajoled can safely leave it out. Using `canEnumOwn` instead will further restrict the enumeration to non-inherited properties, as is typically desired. The same effect can still be obtained more compactly using Cajita's `caja.forEach` construct as explained in section 4.2.

Isolated RegExps. ES3 specifies that a literal regular expression pattern corresponds directly to a single mutable RegExp object. Caja, as well as the Internet Explorer version of JavaScript (JScript), instead create a new RegExp on each evaluation of a literal pattern, avoiding the implicit sharing of mutable state. For any program already compatible with JScript, this is not an issue.

Permissive constructors In JavaScript, if a constructor is stored in an object's property, and that property is then invoked as a method of the object (without using `new`), the constructor would run with its `this` bound to that object,

which in Caja would violate that object’s encapsulation. Even worse, in JavaScript, if a constructor is called as a function, its `this` would be bound to the global object—which would be a fatal escalation of privilege.

In order for Caja to be both safe and a fail-stop subset of JavaScript, these cases should fail. Instead, in the initial Caja implementation, in these cases the constructor may instead act as if called with `new`. This is safe, but it silently diverges from JavaScript behavior.

Attachment breaks identity Figure 6 and Figure 13 each instantiate two points. Both are in Caja-compliant JavaScript—they work correctly whether cajoled or not. After Figure 6, which is also Cajita-compliant, `ptA.getX===ptB.getX` will always be `false`. Whether cajoled or not, each point instance returns its own unique `getX` function.

By contrast, `ptC.getX===ptD.getX` will be `true` if Figure 13 is run uncajoled, but `false` if cajoled. In uncajoled JavaScript, both operands return the `Point.prototype.getX` method itself. When cajoled, the left operand returns the method as attached to `ptC` whereas the right operand returns the method as attached to `ptD`. This difference in object identity is a genuine Caja gotcha. Caja-compliant programs should avoid testing the object identity of methods. Cajita-compliant programs need not worry.

6 Related Work

6.1 Browser Shield

TODO To be written

6.2 ADsafe

TODO To be written

7 Conclusions

TODO To be written

```
function Shadow(model) {
  this.state_ = model.getState();
  var listener = (function(newState) {
    this.state_ = newState;
  }).bind(this);
  model.addStateListener(listener);
}
Shadow.prototype.getState = function() {
  return this.state_;
};
```

Figure 16: A Caja Inner Attached Method. The method appearing within the `Shadow` constructor is an *inner attached method*. It can only be used when its `this` is bound to the same object as its creating context’s `this`, as the above `bind` call ensures.

8 Acknowledgements

We thank Dirk Balfanz, Bruno Bowden, Jon Bright, Andrea Campi, Doug Crockford, Jed Donnelley, Brendan Eich, Adam Langley, Marcel Laverdet, Kevin Reid, and Graham Spencer.

A Tables

References

- [1] 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.
- [2] ECMA. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, Dec. 1999.
- [3] I. K. S. L. Garfinkel. Bitfrost: the One Laptop per Child Security Model. *Symposium On Usable Privacy and Security*, 2007.

Caja expression	cajoles to ES3 code equivalent to	
<code>with</code>	<code>/* rejected in all positions */</code>	(16)
<code>local__</code>	<code>/* rejected in all positions */</code>	(17)
<code>glob_</code>	<code>/* rejected in all positions */</code>	(18)
<code>glob</code>	<code>__OUTERS___.glob</code>	(19)
<code>this.p__</code>	<code>/* rejected in all positions */</code>	(20)
<code>foo.p_</code>	<code>/* rejected in all positions */</code>	(21)
<code>this.p</code>	<code>___.readProp(this, "p")</code>	(22)
<code>foo.p</code>	<code>___.readPub(foo, "p")</code>	(23)
<code>this[bar]</code>	<code>___.readProp(this, bar)</code>	(24)
<code>foo[bar]</code>	<code>___.readPub(foo, bar)</code>	(25)
<code>bar in this</code>	<code>___.canReadProp(this, bar)</code>	(26)
<code>bar in foo</code>	<code>___.canReadPub(foo, bar)</code>	(27)
<code>for (key in this) {...}</code>	<code>for (key in this) {if (___.canEnumProp(this, key)) {...}}</code>	(28)
<code>for (key in foo) {...}</code>	<code>for (key in foo) {if (___.canEnumPub(foo, key)) {...}}</code>	(29)
<code>this.p = baz</code>	<code>___.setProp(this, "p", baz)</code>	(30)
<code>foo.p = baz</code>	<code>___.setPub(foo, "p", baz)</code>	(31)
<code>this[bar] = baz</code>	<code>___.setProp(this, bar, baz)</code>	(32)
<code>foo[bar] = baz</code>	<code>___.setPub(foo, bar, baz)</code>	(33)
<code>glob = baz</code>	<code>__OUTERS___.glob = baz</code>	(34)
<code>var glob = baz</code>	<code>__OUTERS___.glob = baz</code>	(35)
<code>var glob</code>	<code>__OUTERS___.glob = undefined</code>	(36)
<code>delete this.p</code>	<code>___.deleteProp(this, "p")</code>	(37)
<code>delete foo.p</code>	<code>___.deletePub(foo, "p")</code>	(38)
<code>delete this[bar]</code>	<code>___.deleteProp(this, bar)</code>	(39)
<code>delete foo[bar]</code>	<code>___.deletePub(foo, bar)</code>	(40)
<code>delete glob</code>	<code>___.enforce(delete __OUTERS___.glob, ...)</code>	(41)

Figure 17: Cajoling Property Access. Under the assumption that the Caja runtime environment is as specified, the Caja sanitizer generates cajoled Javascript equivalent to that specified above, but inlined and optimized where possible. The meaning of sanitizing is thereby determined by the specification of these entry points into the Caja runtime library. Where we show cajoled code apparently duplicating an expression, the Caja sanitizer instead introduces temporary variables as needed so that each expression evaluates exactly as many times and in the same order as in the original.

Caja expression	cajoles to ES3 code equivalent to	
<i>/* caja module body */</i>	<code>___loadModule(function(___OUTERS___) {</code> <i>/* cajoled module body */</i> <code>});</code>	(42)
<code>this.m(a...)</code>	<code>___callProp(this,"m",[a...])</code>	(43)
<code>foo.m(a...)</code>	<code>___callPub(foo,"m",[a...])</code>	(44)
<code>this[bar](a...)</code>	<code>___callProp(this,bar,[a...])</code>	(45)
<code>foo[bar](a...)</code>	<code>___callPub(foo,bar,[a...])</code>	(46)
<code>new foo(a...)</code>	<code>new (___asCtor(foo))(a...)</code>	(47)
<code>foo(a...)</code>	<code>___asSimpleFunc(foo)(a...)</code>	(48)
Constructors		
<code>function C(a...) {...this...};</code>	<code>function C(var_args) {return new C.make___(arguments);}</code> <code>function C_init___(a...) {...this... }</code> <code>___splitCtor(C,C_init___);/* move to function start */</code>	(49)
<code>function C(a...) {...this...}</code>	<code>___primFreeze(___splitCtor(function C(var_args) {</code> <code>return new C.make___(arguments);</code> <code>},function(a...) {...this...}))</code>	(50)
Methods		
<code>function(a...) {...this...}</code>	<code>___method(C,function(a...) {...this...})</code>	(51)
Simple – functions		
<code>function F(a...) {...};</code>	<code>function F(a...) {...}</code> <code>___simpleFunc(F);/* move to function start */</code>	(52)
<code>function F(a...) {...}</code>	<code>___primFreeze(___simpleFunc(function F(a...) {...}))</code>	(53)
<code>function(a...) {...}</code>	<code>___primFreeze(___simpleFunc(function(a...) {...}))</code>	(54)
<code>arguments.callee</code>	<i>/* rejected */</i>	(55)
<code>...arguments...</code>	<code>...args_____</code>	(56)
	<code>var args___ = ___args(arguments);/* move to function start */</code>	
<code>/pattern/</code>	<code>new RegExp("pattern")</code>	(57)
<code>/pattern/flags</code>	<code>new RegExp("pattern","flags")/* where flags is [igm] * */</code>	(58)

Figure 18: Cajoling Callers and Calleees. A cajoled Caja module can be loaded/evald once, creating an anonymous plugin-maker function. Each time a plugin-maker is called, it makes a new confined plugin. The use of a terminal “;” is shorthand for testing whether the matching expression is evaluated for effects only, not for its value. For each top level named function or constructor declaration F, we also emit “___OUTERS____.F = ___primFreeze(F);” at the end of the module body. Methods may only appear at the member positions in Figure 19.

Caja expression	Special cases for function names and methods	
	Initializes, doesn't freeze Foo	
Foo.prototype.m = member;	----.setMember(Foo,"m",member);	(59)
Foo.prototype = {...:member,...};	----.setMemberMap(Foo,{...:member,...});	(60)
Foo.m = ...	----.setPub(Foo,"m",...)	(61)
caja.def(Foo,Base)		(62)
caja.def(Foo,Base,{...:member,...},...)		
	An inner method within a method or constructor	
member	----.attach(this,member)	(63)
	Freezes Foo to prevent further initialization	
new Foo(...)		(64)
caja.def(Derived, Foo, ...)		
...Foo...primFreeze(Foo)...	(65)
... instanceof Foo	<i>allow, whether Foo is frozen or not</i>	(66)
Foo = ...	<i>reject assignment to a function name</i>	(67)
var Foo = ...	<i>reject conflicting initialization as well</i>	(68)
	Can only happen if Foo is already frozen	
Foo.call(this,...);	<i>Only at start of Derived, and only if the remaining args have no this.</i>	(69)
Foo.prototype.m	<i>Only within methods of Derived</i>	(70)
	----.attach(this, Foo.prototype.m)	

Figure 19: Cajoling Special Cases. When Foo is the name of a named function or a constructor, then these special cases are checked before the general cajoling rules. At the member positions above, either normal expressions or methods may appear.

Methods of ___	method body
<code>enforce(test,complaint)</code>	<code>if (test) { return true; } throw new CajaRuntimeError(complaint);</code>
<code>canRead(obj,name)</code>	<code>return !!obj[name+"_canRead___"];</code>
<code>canEnum(obj,name)</code>	<code>return !!obj[name+"_canEnum___"];</code>
<code>canCall(obj,name)</code>	<code>return !!obj[name+"_canCall___"];</code>
<code>canSet(obj,name)</code>	<code>return !!obj[name+"_canSet___"];</code>
<code>canDelete(obj,name)</code>	<code>return !!obj[name+"_canDelete___"];</code>
<code>allowRead(obj,name)*</code>	<code>obj[name+"_canRead___"] = true;</code>
<code>allowEnum(obj,name)*</code>	<code>allowRead(obj,name); obj[name+"_canEnum___"] = true;</code>
<code>allowCall(obj,name)*</code>	<code>obj[name+"_canCall___"] = true;</code>
<code>allowSet(obj,name)*</code>	<code>enforce(!isFrozen(obj),...); allowEnum(obj,name); obj[name+"_canSet___"] = true;</code>
<code>allowDelete(obj,name)*</code>	<code>enforce(!isFrozen(obj),...); obj[name+"_canDelete___"] = true; /*other bookkeeping yet to be determined*/</code>
<code>hasOwnProp(obj,name)</code>	<code>/*like the original: obj.hasOwnProperty(name)*/</code>
<code>isJSONContainer(obj)</code>	<code>var constr = directConstructor(obj); return constr === Object constr === Array;</code>
<code>isFrozen(obj)</code>	<code>return hasOwnProp(obj,"___FROZEN___");</code>
<code>primFreeze(obj)*</code>	<code>for (k in obj) { if (endsWith(k,"_canSet___") endsWith(k,"_canDelete___")) { obj[k] = false; } obj.___FROZEN___ = true; if (typeof obj === "function") { primFreeze(obj.prototype); } return obj;</code>
<code>method(constr, meth)</code>	<code>enforce(typeof constr === "function",...); enforce(typeof meth === "function",...); meth.___METHOD_OF___ = constr; return primFreeze(meth);</code>
<code>allowMethod(constr,name)*</code>	<code>method(constr,constr.prototype[meth]); allowCall(constr,name);</code>

Figure 20: Hidden Attributes. These methods handle the concrete representations of object and property attributes. Only methods marked with a * should be called by JavaScript code during initialization of the embedding app to express taming decisions. All objects that are reachable from the ES3 shared environment should be frozen, so that the shared environment is transitively read-only to all Caja code.

Methods of ___	method body
canReadProp(that,name)	if (endsWith(name,"_")) { return false; } return canRead(that,name);
readProp(that,name)	return canReadProp(that,name) ? that[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); /*memoize*/ return true;
readPub(obj,name)	return canReadPub(obj,name) ? obj[name] : undefined;
canEnumProp(that,name)	if (endsWith(name,"_")) { return false; } return canEnum(that,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); /*memoize*/ return true;
canSetProp(that,name)	if (endsWith(name,"_")) { return false; } if (canSet(that,name)) { return true; } return !isFrozen(that);
setProp(that,name,val)	enforce(canSetProp(that,name),...); allowSet(that,name); /*grant*/ return that[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)	enforce(canSetPub(obj,name),...); allowSet(obj,name); /*grant*/ return obj[name] = val;
deleteProp(that,name)	enforce(canDeleteProp(that,name),...); /*XXX Bookkeeping yet to be determined*/ return enforce(delete that[name],...);
deletePub(obj,name)	enforce(canDeletePub(obj,name),...); enforce(isJSONContainer(obj),...); /*XXX Bookkeeping yet to be determined*/ return enforce(delete obj[name],...);
args(original)	return primFreeze(Array.prototype.slice.call...(original,0));

Figure 21: 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>	callable*
	all others in ES3	ok, callable

Figure 22: Taming ES3 Global Non-Constructors. Except for `eval`, all non-constructors specified by ES3 are visible in Caja’s outer environment as immutable objects. Note that `Math.random` is not actually immutable, and therefore neither is `Math` nor Caja’s outer environment itself. We allow it anyway for reasons explained in the text.

- [4] A. H. Karp. Authorization-based access control for the services oriented architecture. *c5*, 0:160–167, 2006.
- [5] M. Košík. Backwater Operating System, 2007. altair.dcs.elf.stuba.sk:60001/mediawiki/upload/2/2b/Backwater.pdf.
- [6] B. Laurie. Safer Scripting Through Pre-compilation. *Security Protocols 13*, LNCS 4631, 2004.
- [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.
- [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/.
- [11] M. Stiegler. Emily, a High Performance Language for Secure Cooperation, 2006. 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/.
- [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.

Global ES3 constructor	Property	Taming
constructor		default ctor
constructor.prototype		hidden
	constructor	hidden
	toString	default method
	toLocaleString	default method
	valueOf	default method
instances	length	default ok
	/*stringified numbers*/	default ok
Object.prototype	hasOwnProperty	handled
	isPrototypeOf	method
	propertyIsEnumerable	handled
	freeze_	added method
Function		hidden
Function.prototype		hidden
	apply	handled
	call	handled
	bind	added method
instances	prototype	hidden
	length	ok
Array.prototype	concat	method
	join	method
	pop	handled
	push	handled
	reverse	handled
	shift	handled
	slice	method
	sort	handled
	splice	handled
	unshift	handled
String	fromCharCode	callable
String.prototype	match	handled
	replace	handled
	search	handled
	split	handled
	all others in ES3	ok, method

Figure 23: 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. A *stringified number* is any `x` for which `x === String(Number(x))`. A *handled* method acts differently when called by cajoled *vs.* uncajoled code. Handled mutating methods like `Array.pop` obey Caja’s mutability constraints.

Global ES3 constructor	Property	Taming
Boolean		ctor
Number	MAX_VALUE	ok
	MIN_VALUE	ok
	NaN	ok
	NEGATIVE_INFINITY	ok
	POSITIVE_INFINITY	ok
Number.prototype	toFixed	method
	toExponential	method
	toPrecision	method
Date		ctor*
	parse	callable
Date.prototype	UTC	callable
	to*String all in ES3	method
	get* all in ES3	method
	set* all in ES3	handled
RegExp.prototype	exec	handled
	test	handled
instances	source	ok
	global	ok
	ignoreCase	ok
	multiline	ok
	lastIndex	ok
Error.prototype	name	ok
	message	ok
*Error	all in ES3	ok
*Error.prototype	all in ES3	ok

Figure 24: 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.