On Contracts and Sandboxes for JavaScript

Matthias Keil, Peter Thiemann
University of Freiburg, Germany

August 6, 2015
Darmstadt, Germany
Motivation

89.8 %

of all web sites use JavaScript\textsuperscript{1}

- Most important client-side language for web sites
- Web-developers rely on third-party libraries
  - e.g. for calendars, maps, social networks

\textsuperscript{1}according to http://w3techs.com/, status of July 2015
NASA finds 'Earth's bigger, older cousin'

By Michael Pearson, CNN
(Updated 1435 GMT (2135 HKT) July 24, 2015 | Video Source: CNN/NASA)

**KEPLER-452B IS 1,400 LIGHT YEARS FROM EARTH**

*(CNN)—NASA said Thursday that its Kepler spacecraft has spotted "Earth's bigger, older cousin": the first nearly Earth-size planet to be found in the habitable zone of a star similar to our own.*

Though NASA can’t say for sure whether the planet is rocky like ours or has water and air, it’s the closest
JavaScript issues

- Dynamic programming language
  - Code is accumulated by dynamic loading
  - e.g. eval, mashups
- JavaScript has no security awareness
  - No namespace or encapsulation management
  - Global scope for variables/ functions
  - All scripts have the same authority

Problems

1. Side effects may cause unexpected behavior
2. Program understanding and maintenance is difficult
3. Libraries may get access to sensitive data
4. User code may be prone to injection attacks
Key challenges of present research

- All-or-nothing choice when including code
- Isolation guarantees noninterference
- Some scripts must have access to the application state or are allowed to change it

Goals

1. Manage untrusted JavaScript Code
2. Control the use of data by included scripts
3. Reason about effects of included scripts
Language-embedded Systems

Shortcomings

- Static verifiers are imprecise because of JavaScript’s dynamic features or need to restrict JavaScript’s dynamic features
- Interpreter modifications guarantee full observability but need to be implemented in all existing engines

- Implemented as a library in JavaScript
- Library can easily be included in existing projects
- All aspects are accessible thought an API
- No source code transformation or change in the JavaScript run-time system is required
Timeline

2011
- **JSConTest**
  - Access Permission Contracts for Scripting Languages

2013
- **TreatJS**
  - Higher-Order Contracts for JavaScript
- **JSConTest2**
  - Efficient Access Analysis Using JavaScript Proxies

2015
- **TreatJS-Sandbox**
  - Transaction-based Sandboxing of JavaScript
- **Temporal Contracts, Lemma Contracts, Invariants**

**Ongoing Work**
Access Permission Contracts for Scripting Languages
- Investigate effects of unfamiliar function
- Type and effect contracts with run-time checking
- Summarizes observed access traces to a concise description
- Effect contracts specifying allowed access paths

### Type and effect contracts

```javascript
/**
 * c (obj, obj) -> any with [x.b, y.a]
 */

function f(x, y)
{
    y.a = 1;
    y.b = 2;  // violation
}
```
Shortcomings of JSConTest

- Implemented by an offline code transformation
  - Partial interposition (dynamic code, `eval`, `with`, ...)  
  - Tied to a particular version of JavaScript
  - Transformation hard to maintain

- Special contract syntax
  - Requires a special JavaScript parser

- Efficiency issues
  - Naive representation of access paths
  - Wastes memory and impedes scalability
JSConTest2

Efficient Access Analysis Using JavaScript Proxies
Redesign and reimplementation of JSConTest based on JavaScript proxies

Advantages

- Full interposition for the full language
  - Including dynamically loaded code and `eval`
- Safe for future language extensions
  - No transformation to maintain
- Runs faster in less memory
  - Efficient representation of access paths
  - Incremental path matching
- Maintenance is simplified
  - No custom syntax for contracts
Contracts on Objects

```javascript
var obj = APC.permit('(a.?+b∗)', {a:{b:5},b:{b:11}});
a = obj.a;  // APC.permit('?', {b:5});
a.b = 3;
```

- **APC** encapsulates JSConTest2
- **permit** wraps an object with a permission. Arguments:
  1. Permission encoded in a string
  2. Object that is protected by the permission
- Contract specifies permitted access paths
  - Last property is readable/ writeable
  - Prefix is read-only
  - Not addressed properties are neither readable nor writeable
  - Read-only paths possible (@ denotes a non-existing property)
Proxy Membrane

Contract: $C$
Path: $P$

Contract: $\partial_x C$
Path: $P \cdot x$

Contract: $(\partial_x C) \land (\partial_y C)$
Path: $P \cdot (x \cdot y)$

Matthias Keil, Peter Thiemann
On Contracts and Sandboxes
August 6, 2015 14 / 43
The JSConTest2 Approach

- Implementation based on the JavaScript Proxy API
- Shortcomings of previous, translation-based implementation avoided
  - Full interposition of contracted objects
    - Proxy intercepts all operations
    - Proxy-handler contains a contract and a path set
    - Forwards the operation or signals a violation
  - Returned object contains the remaining contract (Membrane)
- Access contracts are regular expressions on literals
  - Each literal defines a property access
  - The language defines a set of permitted access paths
Language embedded contract system for JavaScript
Enforced by run-time monitoring
Specifies the interface of a software component
Pre- and postconditions
Standard abstractions for higher-order-contracts (base, function, and dependent contracts) [Findler, Felleisen’02]
Systematic blame calculation
Side-effect free contract execution
Contract constructors generalize dependent contracts
Base Contracts [Findler,Felleisen’02]

- Base Contracts are built from predicates
- Specified by a plain JavaScript function

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

assert(1, _Number_); ✓
assert('a', _Number_); ✗ blame the subject
```

- Subject $v$ gets blamed for Base Contract $\mathcal{B}$ iff:
  $\mathcal{B}(v) \neq true$
// Number × Number → Number
function plus (x, y) {
  return (x + y);
}

var plus = assert(plus, Contract.Function([_Number_, _Number_], _Number_));
Function Contract [Findler, Felleisen’02]

// Number × Number → Number
function plus (x, y) {
  return (x + y);
}

plus(’a’, ’a’); X blame the context

- Context gets blamed for $C \rightarrow C'$ iff:
  Argument $x$ gets blamed for $C$ (as subject)
Function Contract [Findler, Felleisen’02]

// Number × Number → Number
function plusBroken (x, y) {
    return (x > 0 && y > 0) ? (x + y) : 'Error';
}

plusBroken(0, 1); ✗ blame the subject

- Subject f gets blamed for C → C' iff:
  ¬ (Context gets blamed C) ∧ (f(x) gets blamed C')
New!
Function `plus` works for strings, too

Requires to model overloading and multiple inheritances

```javascript
// Number × Number → Number
function plus (x, y) {
  return (x + y);
}

plus('a', 'a'); ✗ blame the context
```
Combinations of Contracts

- No support for arbitrary combination of contracts
- Racket supports and\(c\) and or\(c\)
- Attempt to extend conjunction and disjunction to higher-order contracts
Combinations of Contracts

and/c

- and/c tests any contract
- no value fulfills Number and String at the same time

\((\text{and/c } (\text{Number} \times \text{Number} \rightarrow \text{Number}) \ (\text{String} \times \text{String} \rightarrow \text{String}))\)

function plus (x, y) {
  return (x + y);
}

plus(’a’, ’a’); \times blame the context
Combinations of Contracts

or/c

- or/c checks first-order parts and fails unless exactly one (range) contract remains
- Work for disjoint base contracts
- No combination of higher-order contracts
- No support for arbitrary combinations of contracts

\[(or/c (Number \times Number \rightarrow Number) (String \times String \rightarrow String))\]

function plus (x, y) {
    return (x + y);
}

plus('a', 'a'); ✓
Combinations of Contracts

TreatJS

- Support for arbitrary combination of contracts
  - Combination of base and function contracts
  - Combination of function contracts with a different arity
- Intersection and union contracts
- Boolean combination of contracts
Intersection Contract

// (Number × Number → Number) ∩ (String × String → String)

function plus (x, y) {
    return (x + y);
}

var plus = assert(plus, Contract.Intersection(
    Contract.Function([_Number_, _Number_], _Number_)
    Contract.Function([_String_, _String_], _String_));
Intersection Contract

```plaintext
// (Number × Number → Number) ∩ (String × String → String)
function plus (x, y) {
    return (x + y);
}

plus(true, true); X blame the context
```

- Context gets blamed for $C \cap C'$ iff:
  
  (Context gets blamed for $C$) ∧ (Context gets blamed for $C'$)

Matthias Keil, Peter Thiemann
On Contracts and Sandboxes
August 6, 2015 26 / 43
Intersection Contract

// (Number × Number → Number) ∩ (String × String → String)
function plusBroken (x, y) {
    return (x>0 && y>0) ? (x + y) : 'Error';
}

plusBroken(0, 1); ✗ blame the subject

- Subject $f$ gets \textit{blamed} for $C \cap C'$ iff:
  $(f$ gets \textit{blamed} for $C) \lor (f$ gets \textit{blamed} for $C')$
Contract Assertion

- A failing contract must not signal a violation immediately
- Violation depends on combinations of failures in different sub-contracts

```plaintext
// (Number → Number) ∩ (String → String)
function addOne (x) {
    return (x + 1);
}

addOne('a');
```
Contract Assertion

- A failing contract must not signal a violation immediately.
- Violation depends on combinations of failures in different sub-contracts.

```javascript
// (Number → Number) ∩ (String → String)
function addOne (x) {
    return (x + 1);
}

addOne('a');
```
Contract Assertion

- A failing contract must not signal a violation immediately.
- Violation depends on combinations of failures in different sub-contracts.

```
// (Number → Number) ∩ (String → String)
function addOne (x) {
    return (x + 1);
}

addOne('a'); ✓
```
Contract assertion must connect each contract with the enclosing operations

Callback implements a constraint and links each contracts to its next enclosing operation

Reports a record containing two fields, context and subject

Fields range over $\mathbb{B}_4 = \{\bot, f, t, \top\}$ [Belnap’1977]
Non-Interference

- No syntactic restrictions on predicates
- Problem: Contract may interfere with program execution
- Solution: Predicate evaluation takes place in a sandbox

```javascript
function isNumber (arg) {
    type = (typeof arg);
    return type === 'number';
};

var _Number_ = Contract.Base(isNumber);
```
Non-Interference

- No syntactic restrictions on predicates
- Problem: Contract may interfere with program execution
- Solution: Predicate evaluation takes place in a sandbox

```javascript
function isNumber (arg) {
    type = (typeof arg); \* access forbidden
    return type === 'number';
}

var _Number_ = Contract.Base(isNumber);

assert(1, _Number_);
```
- All contracts guarantee noninterference
- Read-only access is safe

```javascript
var _Array_ = Contract.Base(function (arg) {
    return (arg instanceof Array); // access forbidden
});
```
All contracts guarantee noninterference

Read-only access is safe

```javascript
var _Array_ = Contract.Base(function (arg) {
    return (arg instanceof OutsideArray); ✓
});

var _Array_ = Contract.With({OutsideArray:Array}, _Array_);
```
Contract Constructor

- Building block for dependent, parameterized, abstract, and recursive contracts
- Constructor gets evaluated in a sandbox, like a predicate
- Returns a contract
- No further sandboxing for predicates

```javascript
var __Type__ = Contract.Constructor(function (type) {
    return Contract.Base(function (arg) {
        return (typeof arg) === type;
    });
});

var __Number__ = __Type__('number');
```
TreatJS-Sandbox

Transaction-based Sandboxing of JavaScript
TreatJS-Sandbox

- Language-embedded sandbox for full JavaScript
- Inspired by JSConTest2 and Revocable References
- Adapts SpiderMonkey’s compartment concept to run code in isolation to the application state
- Provides features known from transaction processing in database systems and transactional memory
Sandbox Encapsulation

- A reference is the right to access an object
- Requires to control property read and property write

1. Place a write protection on objects
2. Remove external bindings of functions
JavaScript Proxies

```javascript
handler.get(target, 'x', proxy);
handler.set(target, 'y', 1, proxy);
handler.get(target, 'y', proxy);

target['x'];
target['y']=1;
target['y'];
```
Shadow Objects

handler.get(target, 'x', proxy);
handler.set(target, 'y', 1, proxy);
handler.get(target, 'y', proxy);

target['x'];
shadow['y']=1;
shadow['y'];

proxy.x;
proxy.y=1;
proxy.y;
Function decompilation uses the `Function.prototype.toString` method to return a string that contains the source code of that function.

- Applying `eval` to the string creates a fresh variant.
- A `with` statement places a proxy in top of the scope chain.
- The `hasOwnProperty` trap always returns true.
JavaScript Scope Chain

```javascript
var x = 1;

function f (y){
    function g () {
        var z = 1;
        return x+y+z;
    }
    }

Matthias Keil, Peter Thiemann
On Contracts and Sandboxes
August 6, 2015 39 / 43
var $x = 1$;

\begin{verbatim}
with(sbxglobal) {
  function g () {
    var $z = 1$;
    return $x + y + z$;
  }
}
\end{verbatim}
Conclusion

- **JSConTest/ JSConTest2**: Effect monitoring for JavaScript
  Enables to specify effects using access permission contracts

- **TreatJS**: Language embedded, dynamic, higher-order contract system for full JavaScript
  Support for intersection and union contracts
  Contract constructors with local scope

- **Sandbox**: Language embedded sandbox for full JavaScript
  Runs code in a configurable degree of isolation
  Provides a transactional scope
Ongoing Work

- Temporal/Computation Contracts
- Lemma Contracts
- Invariants
- Different blaming semantics (Lax, Picky, Indy)
Further Challenges

Limitations

- Dynamic contract checking impacts the execution time
- Arbitrary combinations of contracts lead to unprecise error messages

1. Hybrid contract checking
2. Static pre-checking of contracts
3. Optimization, contract rewriting