Motivation

89.8 %

of all web sites use JavaScript\(^1\)

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

\(^1\) according to http://w3techs.com/, status of July 2015
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 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 through an API
- No source code transformation or change in the JavaScript run-time system is required
Timelines:

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

- **TreatJS**
  - Higher-Order Contracts for JavaScript
  - 2013

- **Ongoing Work**
  - Temporal Contracts, Lemma Contracts, Invariants
  - ?

---

**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
// < (obj, obj) -> any with [x,b,y.a] ->
function f(x, y) {
  y.b = 1;  // violation
  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

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

```
var obj = APC.permit('(a.?+b∗)′, (a.(b:?),b.(b:11)));
a = obj.a; // APC.permit(′?, (b:?));
```

- 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/ writable
  - Prefix is read-only
  - Not addressed properties are neither readable nor writable
  - Read-only paths possible (Ø denotes a non-existing property)
- Matthias Keil, Peter Thiemann
- On Contracts and Sandboxes
- August 6, 2015

Proxy Membrane

```
Contract: C
Path: P
```

```
Contract: ∂x C
Path: P.x
```

```
Contract: (∂z ∂x C) & (∂y C)
Path: P.(x.z | y)
```

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
- Matthias Keil, Peter Thiemann
- On Contracts and Sandboxes
- August 6, 2015
Introduction

- 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 Contract [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);
```

Subject v gets blamed for Base Contract B iff:

\[ B(v) \neq \text{true} \]
### Function Contract [Findler,Felleisen’02]

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

```javascript
var plus = assert(plus, Contract.Function([Number, _Number, _Number]));
```

---

### Function Contract [Findler,Felleisen’02]

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

```javascript
plusBroken(0, 1);
```

---

Notizen

---

Notizen

---

Notizen

---

Notizen
New!

Overloaded Operator

- 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

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

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

`plus('a', 'a');` blame the context

---

**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-oder contracts
- No support for arbitrary combinations of contracts

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

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

`plus('a', 'a');` /

---

**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]));

plus(true, true);

blame the context

Context gets blamed for \( C \cap C' \) iff:
(Context gets blamed for \( C \)) \land (Context gets blamed for \( C' \))

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 blamed for \( C \cap C' \) iff:
(\( f \) gets blamed for \( C' \)) \lor (\( f \) gets blamed for \( C' \))
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');
```
Blame Calculation

- Contract assertion must connect each contract with the enclosing operations.
- Callback implements a constraint and links each contract to its next enclosing operation.
- Reports a record containing two fields, context and subject.
- Fields range over $B_4 = \{\bot, f, t, \top\}$ [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);
  return type === 'number';
};

var Number = Contract.Base(isNumber);
assert(1, Number);
```
Sandbox

- All contracts guarantee noninterference
- Read-only access is safe

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

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');
```
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

Sandbox Encapsulation
- Place a write protection on objects
- Remove external bindings of functions
Identity Preserving Membrane

JavaScript Proxies

Shadow Objects
Function Recompilation

- 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;
    }
}
```

Sandbox Scope Chain

```javascript
var x = 1;

with(sandbox)
{
    function g ()
    {
        var z = 1;
        return x+y+z;
    }
}
```
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

- Hybrid contract checking
- Static pre-checking of contracts
- Optimization, contract rewriting