October 30, 2016

Virtual values for taint and information flow analysis

Prakasam Kannan, *San Jose State University*
Thomas H. Austin, *San Jose State University*
Mark Stamp, *San Jose State University*
Tim Disney, *Shape Security*
Cormac Flanagan, *University of California, Santa Cruz*

Virtual Values for Taint and Information Flow Analysis

Prakasam Kannan
Thomas H. Austin
Mark Stamp
San José State University
kprakasam@gmail.com /
thomas.austin@sjsu.edu /
stamp@cs.sjsu.edu

Tim Disney
Shape Security
tim.disney@gmail.com

Cormac Flanagan
University of California, Santa Cruz
cormac@ucsc.edu

Abstract

Security controls such as taint analysis and information flow analysis can be powerful tools to protect against many common attacks. However, incorporating these controls into a language such as JavaScript is challenging. Native implementations require the support of all JavaScript VMs. Code rewriting requires developers to reason about the entire abstract syntax of JavaScript.

In this paper, we demonstrate how *virtual values* may be used to more easily integrate these security controls. Virtual values provide hooks to alter the behavior of primitive operations, allowing programmers to create the desired security controls in a more declarative fashion, facilitating more rapid prototyping.

We demonstrate how virtual values may be encoded in JavaScript using a combination of JavaScript object proxies and the Sweet.js macro library, and use that implementation to build taint and information flow controls into JavaScript. Finally, we show some benchmark results to demonstrate the overhead of this approach.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Operating Systems*]: Security and Protection—Information Flow Controls

Keywords virtual values, macros, proxies, taint analysis, information flow analysis

1. Introduction

Taint analysis is a powerful mechanism for preventing code injection attacks. By tracking the flow of untrusted information, we can prevent its use in sensitive operations. For instance, we might require that data entered into a web form must be sanitized before it is used with `eval` or as part of a SQL query. *Information flow analysis* is a stronger extension of taint analysis that protects against *data exfiltration*, when secret data is leaked to an unauthorized viewer.

Despite the power of these mechanisms, adoption has been slow, in part because language designers must integrate these controls into their runtime or their compilation process.

In this paper, we show how *virtual values* [4] can be used by application developers to include taint or information flow controls without requiring support from the underlying JavaScript VM. Virtual values allow the application developer to change the behavior of primitive operations; using this mechanism, developers can instrument their code to track the flow of either tainted or confidential data.

We integrate virtual values into JavaScript using the Sweet.js macro library [14, 15] and JavaScript proxies [10]. JavaScript's proxies allow for behavioral intercession for objects, but do not offer the same support for primitive values. Sweet.js macros allow us to convert primitive values into JavaScript proxies with the additional behavioral hooks needed for virtual values. (Virtual values retain the hooks needed for object proxies, since primitive values can behave like objects).

While our results show a high overhead for virtual values used in this manner, our approach allows developers to include useful additions to the language without relying on support in the underlying VM. Techniques to optimize the performance of metaprogramming such as the use of dispatch chains [26] could reduce this overhead. Additionally, if virtual values are shown to be useful, they could be implemented natively in the JavaScript VMs, which might further improve their performance.

2. Virtual Values Using Sweet.js

JavaScript, as of the ES2015 standard [16], provides a powerful metaprogramming feature called object proxies [10] that allow intercession on all of the standard operations for JavaScript objects. In typical use, a proxy wraps an object, mediating access to that object and changing its behavior. *Traps* are functions on a *handler* object that dictate the be-

Meta'16 October 30, 2016, Amsterdam, Netherlands

```
var vvalues = (function() {
    var unproxyMap = new WeakMap();
    function ValueShell(value) {this.value = value;}
    ValueShell.prototype.valueOf = function() {
        return this.value;
    }
    var oldProxy = this.Proxy;
    this.Proxy = function VProxy(value, handler, key) {
        var valueShell = new ValueShell(value);
        var val = (value == null || typeof value !== 'object') ? valueShell : value;
        var p = new oldProxy(val, handler)
        unproxyMap.set(p, {
            handler: handler,
            key: key,
            target: val
        });
        return p;
    }
    function isVProxy(value) {
        return value && typeof value === 'object' && unproxyMap.has(value);
    }
    function unary(operator, operand) {
        if (isVProxy(operand)) {
            var target = unproxyMap.get(operand).target;
            return unproxyMap.get(operand).handler.unary(target, operator, operand);
        } else if (operator === "-") {
            return -operand;
        } /*** ADDITIONAL UNARY OPERATORS REDACTED FOR SPACE ***/
    }
    function binary(operator, left, right) {
        if (isVProxy(left)) {
            var target = unproxyMap.get(left).target;
            return unproxyMap.get(left).handler.left(target, operator, right);
        } else if (isVProxy(right)) {
            var target = unproxyMap.get(right).target;
            return unproxyMap.get(right).handler.right(target, operator, left);
        } else if (operator === "*") {
            return left * right;
        } /*** ADDITIONAL BINARY OPERATORS REDACTED FOR SPACE ***/
    }
    function assign(left, right, assignThunk) {
        if (isVProxy(left) || isVProxy(right)) {
            return unproxyMap.get(left).handler.assign(left, right, assignThunk);
        } else {
            return assignThunk();
        }
    }
    function test(cond, branchExit) {
        if (isVProxy(cond)) {
            return unproxyMap.get(cond).handler.test(cond, branchExit);
        }
        return cond;
    }
    this.unproxy = function(value, key) {
        if (isVProxy(value) && unproxyMap.get(value).key === key)
            return unproxyMap.get(value).handler;
        return null;
    };
    return {
        unary:  unary,
        binary: binary,
        assign: assign,
        test:   test
    };
})()
```

Figure 1. Virtual Values Harness

havior of the proxy. A wide variety of traps exist [27], such as for getting, setting, or deleting properties from an object.

While JavaScript Proxies are a powerful tool for introducing new behavior to JavaScript objects, they unfortunately cannot extend the behavior of primitive values (e.g. numbers, strings, and booleans).

Virtual values [4] are a proposed extension to object proxies that add support for primitive values to proxies by adding additional traps. This extension includes five additional hooks:

- `unary` - for unary operations.
- `left` - for binary operations, where the left operand is a virtual value.
- `right` - for binary operations, where the right operand is a virtual value.
- `test` - for cases where a virtual value is used as part of a condition.
- `assign` - for assignment operations involving virtual values.

Virtual values have not been added to JavaScript but they can be added via code rewriting, which we do in this paper by using Sweet.js [14, 15], a hygienic macro system for JavaScript. Sweet.js allows us to rewrite the primitive operators in JavaScript (e.g. `+`, `*`, etc.) into the appropriate `unary`, `left`, and `right` function calls. A harness invokes a trap if an operand is a virtual value proxy, or performs the standard JavaScript operation when the value is a primitive.

Figure 1 shows the harness code for creating virtual values. It decorates the `Proxy` object with support for primitive values. The primitive value is wrapped in an instance of the `ValueShell` object, which is then treated as a standard proxy. A mapping of the proxies to their handlers is maintained, allowing the handler for an object to be retrieved via the `unproxy` function. A `key` object is used to allow proxies to recognize themselves.

Operators specify the behavior for the virtual values. If an operand for a unary operator is a virtual value (determined by the `isVProxy` function), then the original value and the handler for the value are retrieved from `unproxyMap`. The `unary` function from the handler is then applied to the target, the operator, and the operand. Binary operators are handled in a similar manner by the `binary` function, though the code is a little more complex. If the left operand is a virtual value, the `left` handler for that value is used. If the left operand is a normal value and the right operand is a virtual value, then the `right` trap of the right operand is invoked. Otherwise, the normal binary operation is applied.

Sweet.js macros allow the default behavior for operators to be overridden. Using Sweet.js macros, all operators are rewritten to use virtual values instead. (If an operator is not specified for a virtual value, using that operator would cause program execution to crash. A possible improvement for this API would be for the standard behavior to be used instead, in a manner similar to the design of JavaScript proxies.) The following code shows the macros for handling the unary operators `!` and `-`, and the binary operators `*` and `/`. In the example below, “13” and “14” specify the precedence of the operator and `left` indicates that an operator is left associative. The template for the generated code is specified by the `#{ ... }` syntax.

```
operator ! 14 { $op }   =>  #{
  vvalues.unary("!", $op)
}
operator - 14 { $op }   =>  #{
  vvalues.unary("-", $op)
}

operator * 13 left { $left, $right }  => #{
  vvalues.binary("*", $left, $right)
}
operator / 13 left { $left, $right }  => #{
  vvalues.binary("/", $left, $right)
}
```

We include support for tracking program influences through the `test` and `assign` hooks. While these hooks are not needed for many use cases, we use it in Section 4 to track leaks from the control flow of a program, generally known as *implicit flows*. We speculate that the same extension could be useful for encoding symbolic execution and other more elaborate tools.

2.1 Performance overhead

In order to better understand the baseline for our system, we modified the popular SunSpider JavaScript performance benchmark [35] to include virtual values. We chose the SunSpider benchmark, as it is focuses on a wide range of JavaScript features from Date, String, and Regexp manipulation to a wide variety of numerical, array-oriented, object-oriented, and functional idioms. No other changes were done to the benchmark, and the virtual values in these tests pass through all operations without otherwise changing behavior, allowing us to establish the baseline overhead of virtual values alone.

These tests were run on a Mac Book Pro with one 2.6 GHz Intel Core i7 processor containing 4 cores, 16 GB of RAM, and an Intel Iris Pro graphics processor with 1536 MB of memory. We used the Sweet.js compiler version 0.7.8 to translate version 1.02 of the SunSpider benchmark. Three tests cases (3d/raytrace, crypto/aes, date/format-tofte) were excluded from the testing since they contain minified JavaScript that made modification difficult. The resulting code was tested on Safari, Chrome, and Firefox.

Table 1 shows the results of our testing. In all cases, virtual values introduce significant overhead. Interestingly, though Safari performed best without Sweet.js, Chrome’s results were best on the Sweet.js-compiled code.

Rewriting JavaScript operations into function calls comes with a certain performance penalty. Despite the significant overhead, it is not atypical for code-rewriting approaches [8, 9]. We are hopeful that future version of JavaScript might one day support virtual values natively, eliminating the cost of introducing virtual values.

For future work, we plan to augment the Sweet.js virtual value compiler to identify expressions that do not involve proxies during the parse phase and avoid the rewriting operations into function calls.

3. Taint Analysis

Taint analysis is a language feature that tracks and restricts the flow of data through a program. Taint analysis is accomplished by programmers indicating which inputs should be tracked and which outputs should not accept tainted values. This prevents common programming mistakes such as failing to sanitize user input. Previous research has used taint tracking to detect application vulnerabilities [29, 37], and it is a built-in feature of languages such as Perl and Ruby.

While taint analysis is not currently available in JavaScript, the browser is a rich setting for all number of potentially unsafe inputs that could benefit from taint analysis. As one example, we might wish to prevent a string taken from a form element from being passed to `eval`. By tracking this information, we can allow it to be used freely up until the point where it might be used in an unsafe manner.

3.1 Taint Analysis API

Our JavaScript API for taint analysis consists of three functions provided to the programmer: `taint`, `isTainted`, and `endorse`. The `taint` function takes a value and taints it, the `isTainted` function takes a value and returns `true` if the value is tainted, and the `endorse` function removes the taint from a value. The following code shows the use of this API:

```
var username =
    taint("Robert'); DROP TABLE Students;--");
var query  = "select * from Students " +
    "where username = '" + username + "'");
if (isTainted(query))
    throw new Error("Tainted query");
```

Note that a tainted value must be propagated through primitive operations that create new values. In the above example the concatenation of `username` with other strings must result in `query` being tainted as well.

Leveraging object proxies and virtual values, the code required to implement `taint` and `isTainted` is pleasingly minimal. Figure 2 shows the required functions to introduce taint analysis controls.

The `taint` function wraps a value inside a virtual value where the `unary`, `left`, and `right` hooks propagate the taint onto the result of the computation, performed by applying functions in the `unaryOps` and `binaryOps` arrays. The `unaryOps` and `binaryOps` objects map symbols to functions performing the default behavior for the given operator.

The `taintingKey` used in the `taint` function allows the `isTainted` function to detect when a value is tainted. It also is used to retrieve the original, untainted value of a virtual value using the `endorse` function.

```
// this object is used to identify proxies
// crated by the `taint` function
var taintingKey = {};

function taint(originalValue) {
  if (isTainted(originalValue)) {
    return originalValue;
  }
  var p = new Proxy(originalValue, {
    // Store the original untainted
    // value for later.
    originalValue: originalValue,
    unary: function (target, op, operand) {
      return taint(unaryOps[op](target));
    },
    left: function (target, op, right) {
      return taint(binaryOps[op](target,
                                 right));
    },
    right: function (target, op, left) {
      return taint(binaryOps[op](left,
                                 target));
    }
  }, taintingKey);
  return p;
}

function isTainted (x) {
  // a value is tainted if it is a proxy
  // created with the 'taintingKey'
  if (unproxy(x, taintingKey)) {
    return true;
  }
  return false;
}

function endorse (value) {
  if (isTainted(value)) {
    // pulls the value out of
    // its tainting proxy
    return unproxy(value,
        taintingKey).originalValue;
  }
  return value;
}
```

Figure 2. Taint Analysis Functions

3.2 Performance Tests for Taint Tracking

While Table 1 shows the baseline overhead of virtual values, we also wish to evaluate the overhead of leveraging virtual values to implement security controls.

We use the `validate-input` test case in Sun Spider to determine the additional overhead introduced by taint tracking. 4000 email addresses and zip codes are generated and validated using regular expressions. We tainted a portion of these email addresses and zip codes. Table 2 shows the results; while virtual values add significant performance overhead, using them for taint analysis adds comparatively little additional load. Despite an exponential increase in the amount of tainted variables, the performance overhead increases only slightly.

Test	Safari		Chrome		Firefox	
	Base	Virtual Values	Base	Virtual Values	Base	Virtual Values
3d	10.0ms	73.3ms	18.0ms	75.5ms	17.0ms	80.9ms
cube	5.0ms	31.8ms	8.8ms	33.3ms	12.6ms	44.2ms
morph	5.0ms	41.5ms	9.2ms	42.2ms	4.5ms	36.7ms
access	13.0ms	122.0ms	11.5ms	101.7ms	13.9ms	139.3ms
binary-trees	2.2ms	8.7ms	1.5ms	7.9ms	3.0ms	10.9ms
fannkuch	5.2ms	72.6ms	5.6ms	55.4ms	5.5ms	83.7ms
nbody	2.6ms	23.0ms	2.1ms	23.0ms	2.8ms	21.8ms
nsieve	3.0ms	17.7ms	2.3ms	15.4ms	2.6ms	22.9ms
bitops	9.1ms	159.1ms	18.9ms	126.5ms	7.7ms	222.4ms
3bit-bits-in-byte	1.0ms	30.0ms	1.0ms	25.7ms	0.8ms	46.3ms
bits-in-byte	3.0ms	35.0ms	3.8ms	30.9ms	1.6ms	53.2ms
bitwise-and	2.0ms	28.5ms	11.1ms	25.5ms	2.2ms	43.7ms
nsieve-bits	3.1ms	65.6ms	3.0ms	44.4ms	3.1ms	79.2ms
controlflow	2.1ms	14.4ms	1.3ms	10.6ms	2.0ms	16.0ms
recursive	2.1ms	14.4ms	1.3ms	10.6ms	2.0ms	16.0ms
crypto	5.0ms	42.0ms	7.3ms	41.7ms	6.7ms	62.0ms
md5	2.4ms	21.0ms	3.6ms	20.3ms	3.7ms	30.6ms
sha1	2.6ms	21.0ms	3.7ms	21.4ms	3.0ms	31.4ms
date	5.2ms	9.3ms	11.2ms	15.7ms	11.1ms	33.2ms
format-xparb	5.2ms	9.3ms	11.2ms	15.7ms	11.1ms	33.2ms
math	9.3ms	81.2ms	12.9ms	77.9ms	10.4ms	88.0ms
cordic	3.0ms	40.7ms	3.0ms	36.6ms	2.2ms	46.6ms
partial-sums	4.3ms	15.0ms	7.9ms	21.9ms	6.6ms	18.6ms
spectral-norm	2.0ms	25.5ms	2.0ms	19.4ms	1.6ms	22.8ms
regexp	5.7ms	5.3ms	5.5ms	6.2ms	6.6ms	7.9ms
dna	5.7ms	5.3ms	5.5ms	6.2ms	6.6ms	7.9ms
string	23.7ms	81.5ms	43.8ms	88.5ms	30.9ms	101.9ms
base64	4.3ms	22.2ms	4.2ms	19.2ms	5.7ms	28.8ms
fasta	6.1ms	25.1ms	11.4ms	22.7ms	6.0ms	25.1ms
tagcloud	8.9ms	19.5ms	22.3ms	30.2ms	13.2ms	30.6ms
validate-input	4.4ms	14.7ms	5.9ms	16.4ms	6.0ms	17.4ms
Total	83.1ms	588.1ms	130.4ms	544.3ms	142.3ms	751.6ms
	(7.1x slowdown)		**(4.2x slowdown)**		**(5.3x slowdown)**	

Table 1. SunSpider Performance with Virtual Values

Num. of Variables	Tainted Variables	Time
4000	40	18.6ms
4000	80	19.1ms
4000	160	19.1ms
4000	320	19.2ms
4000	640	19.3ms
4000	1280	19.5ms

Table 2. Taint Performance Test Results

4. Information Flow Analysis

Information flow analysis extends taint analysis to handle confidentiality concerns; that is, it is focused on protecting secret information from being leaked, rather than preventing code injection attacks. Early work on information flow analysis focused on static approaches, such as Denning's strategy of including an information flow certification component in a compiler [11, 12], or information flow type systems [20, 38]. While these techniques have been studied widely for statically typed languages, such as the Java-like Jif language [22, 28] and FlowCaml [30], they seem less fitting for dynamic languages. Dynamic information flow analysis for JavaScript in particular has been the source of significant attention [7, 9, 13, 19, 23, 24, 32, 34].

In addition to the *explicit flows* of information handled in taint analysis, with information flow analysis we must also consider *implicit flows*, where an attacker learns information through the control flow of the program. For a simple exam-

x =	false^H	true^H	
Function f(x)	*Both strategies*	*Naive*	*NSU*
y = true;	y = true	y = true	y = true
z = true;	z = true	z = true	z = true
if (x)	–	$pc = H$	$pc = H$
y = false;	–	y = false^H	*stuck*
if (y)	$pc = L$	–	
z = false;	z = false	–	
return z;	–	–	
Return Value:	false	true	

Figure 3. A JavaScript function with implicit flows

ple, consider the following code with an implicit flow from the secret variable sec to the public output:

```
var sec = secret(true);
var pub = false;
if (sec) {
  pub = true;
}
console.log(pub)
```

Although an attacker cannot observe sec, the public value of pub reveals the value of sec, even though there has been no direct assignment from sec to pub. Unlike taint tracking, information flow analysis assumes that attackers can control some portion of the code, and therefore can build sophisticated implicit flows if they are not tracked correctly.

Implicit flows are surprisingly complex to handle correctly. Figure 4 shows an example to illustrate these challenges, adapted from a code example first discovered by Fenton [17]. We review two strategies: the "naive" strategy marks data as confidential, denoted by the superscript H for "high", whenever it is updated in a sensitive context; the no-sensitive-upgrade strategy [1, 39], given in the NSU column, instead terminates execution when confidential information might be leaked.

If this function is called with a secret false value, denoted false^H, then both approaches handle execution in the same manner. Since x is false^H, y remains true and public. Therefore, z is updated to false in the second conditional, and remains public.

If the function is instead called with true^H, the naive approach tracks the sensitive influence in the first conditional statement by setting the program counter to confidential ($pc = H$), and tracks its influence by setting y to false^H. Therefore, no update to z is performed, and its value remains false and public, thereby leaking one bit of data.

To prevent against this implicit leak, the NSU strategy disallows updates to public references in a confidential context. When y is updated, execution "gets stuck" and terminates the application. More permissive approaches exist for dynamically handling these cases, such as the permissive-upgrade strategy [2, 6], secure multi-execution [13, 21, 31],

```
let key = {};
let pcStack = [];
function secret(originalValue) {
  if (isSecret(originalValue)) {
    return originalValue;
  }
  var p = new Proxy(originalValue, {
    originalValue: originalValue,
    unary: function (target, op, operand) {
      return secret(unaryOps[op](target));
    },
    left: function (target, op, right) {
      return secret(binaryOps[op](target, right));
    },
    right: function (target, op, left) {
      return secret(binaryOps[op](left, target));
    }
    test: function(cond, branchExit) {
      if (cond) {
        pcStack.push(cond);
        branchExit(() => {
          pcStack.pop();
        })
      }
      return cond;
    },
    assign: function(left, right, assignThunk) {
      if (pcStack.length > 0) {
        throw new Error("Implicit leak");
      }
      assignThunk();
    }
  }, key);
  return p;
}

function isSecret (x) {
  return unproxy(x, key);
}
```

Figure 4. Information Flow Functions

and faceted values [5, 33]. We select the NSU approach for illustrative purposes since it is easier to understand.

Using virtual values and Sweet.js macros, we can implement the NSU strategy within JavaScript. To detect implicit flows, we need to maintain a program counter (pc) of influences on the current execution.

Our implementation in Figure 1 provides the appropriate hooks to track the program counter. Tracking the pc is accomplished by extending the test handler (which traps an if statement) with a branchExit registration parameter. The branchExit parameter is a function that takes a callback to be invoked once the if statement's then branch has completed.

The extended test handler allows our implementation of NSU (see Figure 4) to push and pop "influence" (represented by a virtual value) onto a program counter stack.

To prevent implicit flows, the assign handler looks on the program counter stack to see if it is inside of a high security context; if so, it throws an error. To implement the test handler, we use a Sweet.js macro to expand if statements into the appropriate virtual values calls. Macros in Sweet.js use the following form, where <pattern> gives the pattern to match in the input program and <template> gives the pattern of the generated code.

```
macro {
  rule {
    <pattern>
  } => {
    <template>
  }
}
```

The macro for if statements shows how we can change the behavior of control structures to track information flow.

```
macro if {
  rule { ($cond ...) { $body ...} }  => {
    function exit() { } // by default no-op
    if (vvalues.test($cond...,
                     cb => exit = cb)) {
      $body ...
      exit();
    }
  }
}
```

We also need to modify how assignment behaves, which we can do by using Sweet.js *infix* macros. Infix macros allow us to match syntax before the distinguishing identifier.

```
macro = {
  rule infix { $left | $right:expr } => {
    vvalues.assign($left, $right, () => {
      $left = $right
    });
  }
}
```

5. Related Work

The original paper on virtual values [4] gives the hooks necessary to support primitive values in JavaScript. While it only has a proof-of-concept implementation, many interesting use cases are demonstrated. We extend that work with additional features to support more advanced use cases, like information flow analysis. Additionally, we show how virtual values can be encoded into JavaScript through a combination of JavaScript proxies and Sweet.js macros.

JavaScript proxies [10] are closely related to virtual values. Proxies only support operations for objects, making them ineffective for certain types of analysis.

Christophe et al. [8] develop Linvail for JavaScript, providing a general purpose framework for dynamic analysis in JavaScript. This work also demonstrates how taint analysis could be supported, and discusses the challenges of tracking primitive values in JavaScript.

Rewriting code to ensure security guarantees has been used in several domains. Maffeis and Taly [25] explore the guarantees for these tools for JavaScript specifically. Caja [18] uses a "cajoler" that rewrites code to follow the object capabilities model, thereby preventing untrusted code from accessing powerful libraries. Taly et al. [36] formalize a subset of JavaScript and use it to analyze these code rewriting APIs. Chudnov and Naumann [9] rewrite JavaScript code to provide information flow guarantees using the no-sensitive-upgrade approach. The main benefit of our approach is that, once the correct virtual values hooks are available, the security controls can be rewritten in a more declarative approach, without needing to consider the complete abstract syntax of JavaScript.

6. Conclusion and Future Work

In this paper, we have demonstrated how virtual values may be implemented in JavaScript using proxies and Sweet.js macros. We have further shown how taint tracking and information flow analysis can be implemented using virtual values. By showing how these security controls can be implemented within a language using various metaprogramming techniques, we hope to accelerate adoption of security tools.

Sweet.js has recently gone through a major redesign. For future work, we intend to extend our design to work with the latest version of the library, and also to explore how additional security mechanisms such as faceted values [3] can be encoded through virtual values.

References

[1] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Programming Languages and Analysis for Security*, pages 113–124. ACM, 2009. ISBN 978-1-60558-645-8.

[2] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Programming Languages and Analysis for Security*, pages 1–12. ACM, 2010.

[3] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Symposium on Principles of Programming Languages (POPL)*, pages 165–178. ACM, 2012. ISBN 978-1-4503-1083-3.

[4] T. H. Austin, T. Disney, and C. Flanagan. Virtual values for language extension. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 921–938. ACM, 2011.

[5] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted execution of policy-agnostic programs. In *Programming Languages and Analysis for Security*. ACM, 2013.

[6] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Generalizing permissive-upgrade in dynamic information flow analysis. In *Programming Languages and Analysis for Security*. ACM, 2014.

[7] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in webkit's javascript bytecode. In *Principles of Security and Trust (POST)*, pages 159–178. Springer, 2014.

[8] L. Christophe, E. G. Boix, W. D. Meuter, and C. D. Roover. Linvail: A general-purpose platform for shadow execution of javascript. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 260–270. IEEE Computer Society, 2016.

[9] A. Chudnov and D. A. Naumann. Inlined information flow monitoring for javascript. In *Conference on Computer and Communications Security (SIGSAC)*, pages 629–643. ACM,

2015. URL http://doi.acm.org/10.1145/2810103.2813684.

[10] T. V. Cutsem and M. S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In *Dynamic Languages Symposium (DLS)*. ACM, 2010.

[11] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[12] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20 (7):504–513, 1977.

[13] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Symposium on Security and Privacy*, pages 109–124, Los Alamitos, CA, USA, 2010. IEEE.

[14] T. Disney. Sweet.js – sweeten your javascript. http://sweetjs.org/. Accessed: August 2016.

[15] T. Disney, N. Faubion, D. Herman, and C. Flanagan. Sweeten your javascript: hygienic macros for ES5. In *Dynamic Languages Symposium (DLS)*, pages 35–44. ACM, 2014.

[16] ECMA International. *Standard ECMA-262 - ECMAScript 2015 Language Specification*. 6.0 edition, June 2015. URL http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf.

[17] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[18] Google. Google's Caja project. Accessed May 2016 from https://developers.google.com/caja/.

[19] D. Hedin and A. Sabelfeld. Web application security using jsflow. In *17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2015, Timisoara, Romania, September 21-24, 2015*, pages 16–19. IEEE, 2015.

[20] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Symposium on Principles of Programming Languages (POPL)*, pages 365–377. ACM, 1998.

[21] M. Jaskelioff and A. Russo. Secure multi-execution in haskell. In *Ershov Memorial Conference*, pages 170–178. Springer, 2011.

[22] Jif. Jif homepage. http://www.cs.cornell.edu/jif/.

[23] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz. Information flow tracking meets just-in-time compilation. *Transactions on Architecture and Code Optimization (TACO)*, 10(4):38, 2013.

[24] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz. Towards precise and efficient information flow control in web browsers. In *Trust and Trustworthy Computing Conference*. Springer, 2013.

[25] S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *Computer Security Foundations Workshop (CSFW)*, pages 77–91. IEEE, 2009.

[26] S. Marr, C. Seaton, and S. Ducasse. Zero-overhead metaprogramming: reflection and metaobject protocols fast and without compromises. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 545–554. ACM, 2015.

[27] Mozilla Developer Network. Proxy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy. Accessed: August 2016.

[28] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages (POPL)*, pages 228–241. ACM, 1999.

[29] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2005.

[30] F. Pottier and V. Simonet. Information flow inference for ML. In *Symposium on Principles of Programming Languages (POPL)*, pages 319–330. ACM, 2002.

[31] W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *Computer Security Foundations Symposium (CSF)*. IEEE, 2013.

[32] J. F. Santos, T. Jensen, T. Rezk, and A. Schmitt. Hybrid Typing of Secure Information Flow in a JavaScript-Like Language. In *Trustworthy Global Computing Symposium (TGC)*, pages 63–78, 2015.

[33] T. Schmitz, D. Rhodes, T. H. Austin, K. Knowles, and C. Flanagan. Faceted information flow in Haskell via control and data monads. In *Principles of Security and Trust (POST)*. Springer, 2016.

[34] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. Protecting users by confining javascript with COWL. In *Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 131–146. USENIX Association, 2014.

[35] Sun Spider. 1.0.2 JavaScript benchmark. https://webkit.org/perf/sunspider/sunspider.html. Accessed: April 2016.

[36] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. In *Symposium on Security and Privacy (S&P)*, pages 363–378. IEEE Computer Society, 2011.

[37] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 87–97. ACM, 2009.

[38] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3): 167–187, 1996.

[39] S. A. Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.