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
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-
```javascript
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
```
behavior 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.

```javascript
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:

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

if (isTainted(query))
    throw new Error("Tainted query");

function endorse (value) {
    return taint (unaryOps [ op ] (target ,
                                    operand ));
}

function isTainted (value) {
    return originalValue;
}

function taint (originalValue) {
    return originalValue;
}
```

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.

```javascript
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 (binaryOps [ op ] (target ,
                                             operand ));
        },
        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;
}
```

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

```javascript
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.
Table 1. SunSpider Performance with Virtual Values

<table>
<thead>
<tr>
<th>Test</th>
<th>Safari Base</th>
<th>Safari Virtual Values</th>
<th>Chrome Base</th>
<th>Chrome Virtual Values</th>
<th>Firefox Base</th>
<th>Firefox Virtual Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>3d</td>
<td>10.0ms</td>
<td>73.3ms</td>
<td>18.0ms</td>
<td>75.5ms</td>
<td>17.0ms</td>
<td>80.9ms</td>
</tr>
<tr>
<td>cube</td>
<td>5.0ms</td>
<td>31.8ms</td>
<td>8.6ms</td>
<td>33.3ms</td>
<td>12.6ms</td>
<td>44.2ms</td>
</tr>
<tr>
<td>morph</td>
<td>5.0ms</td>
<td>41.5ms</td>
<td>9.2ms</td>
<td>42.2ms</td>
<td>4.5ms</td>
<td>36.7ms</td>
</tr>
<tr>
<td>access</td>
<td>13.0ms</td>
<td>122.0ms</td>
<td>11.3ms</td>
<td>101.7ms</td>
<td>13.9ms</td>
<td>139.3ms</td>
</tr>
<tr>
<td>binary-trees</td>
<td>2.2ms</td>
<td>8.7ms</td>
<td>1.5ms</td>
<td>7.9ms</td>
<td>3.0ms</td>
<td>10.9ms</td>
</tr>
<tr>
<td>fannkuch</td>
<td>5.2ms</td>
<td>72.6ms</td>
<td>5.6ms</td>
<td>55.4ms</td>
<td>5.5ms</td>
<td>83.7ms</td>
</tr>
<tr>
<td>nbody</td>
<td>2.6ms</td>
<td>23.0ms</td>
<td>2.1ms</td>
<td>23.0ms</td>
<td>2.8ms</td>
<td>21.8ms</td>
</tr>
<tr>
<td>nsieve</td>
<td>3.0ms</td>
<td>17.7ms</td>
<td>2.3ms</td>
<td>15.4ms</td>
<td>2.6ms</td>
<td>22.9ms</td>
</tr>
<tr>
<td>bitops</td>
<td>9.1ms</td>
<td>159.1ms</td>
<td>18.9ms</td>
<td>126.5ms</td>
<td>7.7ms</td>
<td>222.4ms</td>
</tr>
<tr>
<td>3bit-bits-in-byte</td>
<td>1.0ms</td>
<td>30.0ms</td>
<td>1.0ms</td>
<td>25.7ms</td>
<td>0.8ms</td>
<td>46.3ms</td>
</tr>
<tr>
<td>bits-in-byte</td>
<td>3.0ms</td>
<td>35.0ms</td>
<td>3.8ms</td>
<td>30.9ms</td>
<td>1.6ms</td>
<td>53.2ms</td>
</tr>
<tr>
<td>bitwise-and</td>
<td>2.0ms</td>
<td>28.5ms</td>
<td>11.1ms</td>
<td>25.5ms</td>
<td>2.2ms</td>
<td>43.7ms</td>
</tr>
<tr>
<td>nsieve-bits</td>
<td>3.1ms</td>
<td>65.6ms</td>
<td>3.0ms</td>
<td>44.4ms</td>
<td>3.1ms</td>
<td>79.2ms</td>
</tr>
<tr>
<td>controlflow</td>
<td>2.1ms</td>
<td>14.4ms</td>
<td>1.3ms</td>
<td>10.6ms</td>
<td>2.0ms</td>
<td>16.0ms</td>
</tr>
<tr>
<td>recursive</td>
<td>2.1ms</td>
<td>14.4ms</td>
<td>1.3ms</td>
<td>10.6ms</td>
<td>2.0ms</td>
<td>16.0ms</td>
</tr>
<tr>
<td>crypto</td>
<td>5.0ms</td>
<td>42.0ms</td>
<td>7.3ms</td>
<td>41.7ms</td>
<td>6.7ms</td>
<td>62.0ms</td>
</tr>
<tr>
<td>md5</td>
<td>2.4ms</td>
<td>21.0ms</td>
<td>3.6ms</td>
<td>20.3ms</td>
<td>3.7ms</td>
<td>30.6ms</td>
</tr>
<tr>
<td>sha1</td>
<td>2.6ms</td>
<td>21.0ms</td>
<td>3.7ms</td>
<td>21.4ms</td>
<td>3.0ms</td>
<td>31.4ms</td>
</tr>
<tr>
<td>date</td>
<td>5.2ms</td>
<td>9.3ms</td>
<td>11.2ms</td>
<td>15.7ms</td>
<td>11.1ms</td>
<td>33.2ms</td>
</tr>
<tr>
<td>format-xparb</td>
<td>5.2ms</td>
<td>9.3ms</td>
<td>11.2ms</td>
<td>15.7ms</td>
<td>11.1ms</td>
<td>33.2ms</td>
</tr>
<tr>
<td>math</td>
<td>9.3ms</td>
<td>81.2ms</td>
<td>12.9ms</td>
<td>77.9ms</td>
<td>10.4ms</td>
<td>88.0ms</td>
</tr>
<tr>
<td>cordic</td>
<td>3.0ms</td>
<td>40.7ms</td>
<td>3.0ms</td>
<td>36.6ms</td>
<td>2.2ms</td>
<td>46.6ms</td>
</tr>
<tr>
<td>partial-sums</td>
<td>4.3ms</td>
<td>15.0ms</td>
<td>7.9ms</td>
<td>21.9ms</td>
<td>6.6ms</td>
<td>18.6ms</td>
</tr>
<tr>
<td>spectral-norm</td>
<td>2.0ms</td>
<td>25.5ms</td>
<td>2.0ms</td>
<td>19.4ms</td>
<td>1.6ms</td>
<td>22.8ms</td>
</tr>
<tr>
<td>regexp</td>
<td>5.7ms</td>
<td>5.3ms</td>
<td>5.5ms</td>
<td>6.2ms</td>
<td>6.6ms</td>
<td>7.9ms</td>
</tr>
<tr>
<td>dna</td>
<td>5.7ms</td>
<td>5.3ms</td>
<td>5.5ms</td>
<td>6.2ms</td>
<td>6.6ms</td>
<td>7.9ms</td>
</tr>
<tr>
<td>string</td>
<td>23.7ms</td>
<td>81.5ms</td>
<td>43.8ms</td>
<td>88.5ms</td>
<td>30.9ms</td>
<td>101.9ms</td>
</tr>
<tr>
<td>base64</td>
<td>4.3ms</td>
<td>22.2ms</td>
<td>4.2ms</td>
<td>19.2ms</td>
<td>5.7ms</td>
<td>28.8ms</td>
</tr>
<tr>
<td>fasta</td>
<td>6.1ms</td>
<td>25.1ms</td>
<td>11.4ms</td>
<td>22.7ms</td>
<td>6.0ms</td>
<td>25.1ms</td>
</tr>
<tr>
<td>tagcloud</td>
<td>8.9ms</td>
<td>19.5ms</td>
<td>22.3ms</td>
<td>30.2ms</td>
<td>13.2ms</td>
<td>30.6ms</td>
</tr>
<tr>
<td>validate-input</td>
<td>4.4ms</td>
<td>14.7ms</td>
<td>5.9ms</td>
<td>16.4ms</td>
<td>6.0ms</td>
<td>17.4ms</td>
</tr>
<tr>
<td>Total</td>
<td>83.1ms</td>
<td>588.1ms</td>
<td>130.4ms</td>
<td>544.3ms</td>
<td>142.3ms</td>
<td>751.6ms</td>
</tr>
</tbody>
</table>

(7.1x slowdown) (4.2x slowdown) (5.3x slowdown)

Table 2. Taint Performance Test Results

<table>
<thead>
<tr>
<th>Num. of Variables</th>
<th>Tainted Variables</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>40</td>
<td>18.6ms</td>
</tr>
<tr>
<td>4000</td>
<td>80</td>
<td>19.1ms</td>
</tr>
<tr>
<td>4000</td>
<td>160</td>
<td>19.1ms</td>
</tr>
<tr>
<td>4000</td>
<td>320</td>
<td>19.2ms</td>
</tr>
<tr>
<td>4000</td>
<td>640</td>
<td>19.3ms</td>
</tr>
<tr>
<td>4000</td>
<td>1280</td>
<td>19.5ms</td>
</tr>
</tbody>
</table>

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-
### Table

<table>
<thead>
<tr>
<th>x =</th>
<th>false&lt;sup&gt;H&lt;/sup&gt;</th>
<th>true&lt;sup&gt;H&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function f(x)</td>
<td>Both strategies</td>
<td>Naive</td>
</tr>
<tr>
<td>y = true;</td>
<td>y = true</td>
<td>y = true</td>
</tr>
<tr>
<td>z = true;</td>
<td>z = true</td>
<td>z = true</td>
</tr>
<tr>
<td>if (x)</td>
<td>pc = H</td>
<td>pc = H</td>
</tr>
<tr>
<td>y = false;</td>
<td>y = false&lt;sup&gt;H&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>if (y)</td>
<td>pc = L</td>
<td>-</td>
</tr>
<tr>
<td>z = false;</td>
<td>z = false</td>
<td>-</td>
</tr>
<tr>
<td>return z;</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Return Value:</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

### Figure 3

A JavaScript function with implicit flows

```javascript
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();
    },
};

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

### Figure 4

Information Flow Functions

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 static 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 non-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<sup>H</sup>, then both approaches handle execution in the same manner. Since x is false<sup>H</sup>, y remains true and public. Therefore, z is updated to false<sup>H</sup> in the second conditional, and remains public.

If the function is instead called with true<sup>H</sup>, 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<sup>H</sup>. 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], and facetted 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 \(<\text{pattern}\>\) gives the pattern to match in the input program and \(<\text{template}\>\) gives the pattern of the generated code.

```javascript
macro {
  rule {
    \(<\text{pattern}\>\) \=> {
      \(<\text{template}\>\)
    }
  }
}
```

The macro for \(\text{if}\) statements shows how we can change the behavior of control structures to track information flow.

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

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

```javascript
macro = {
  rule infix { \$left \mid \$right: \text{expr} } \=> {
    \text{vvalues\text{.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


