ABSTRACT
In this paper we have implemented a fairly robust taint tracking facility in the JavaScript language implementation of the Microsoft Research’s C3 system. We have also implemented a comprehensive suite of test cases (in JavaScript) along with a framework (in C#) ensuring that our semantics have been correctly implemented. Using our taint tracking system, we have illustrated a proof of concept test case that enables us to track the flow of taint in the DOM (effectively the browser).

1. INTRODUCTION
Over the past decade computing has shifted increasingly towards web-based applications and services, affording more opportunities for malicious exploits aimed at stealing user-sensitive information such as passwords, credit card information and Social Security Numbers. Domain-crossing exploits such as cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks are commonly employed by modern ne’er-do-wells to steal others’ personal data and identities. Recently, dynamic taint tracking has been shown effective in sniffing-out such exploits. However, the components of many modern web browsers - the layout engine, Document Object Model (DOM)\(^1\), JavaScript handler and HTML and CSS parser - are implemented in different languages and styles which complicates taint tracking across the entire browser. Furthermore, the dynamic, prototype-based nature of the JavaScript language complicates the implementation of dynamic taint tracking. Microsoft Research resolves difficulties tied to heterogeneous browser implementations with their new C3 Web browser. Designed as a client product for accessing cloud-computing resources, all of the C3 browser’s subsystems are integrated within the common .NET framework. We extend the C3 browser to create TaintSNIFFER, a dynamic taint tracking system for C3’s JavaScript engine and DOM subsystem. We author a suite of correctness benchmarks for validating implementations of dynamic taint tracking in JavaScript and web browsing systems in general, then use TaintSNIFFER to verify the suite’s robustness and practicality.

The remainder of this paper is organized as follows. Section 2 describes our approach in technical detail. Section 3 presents relevant related work of others in leveraging dynamic taint tracking for preventing malicious attacks in Web browsers, specifically in their JavaScript engines. Section 4 presents our experimental methodology for evaluating our dynamic taint tracking system. In Section 5 we present and discuss the empirical results obtained during our evaluation. We discuss the limitations of TaintSNIFFER and our test suite in Section 6. Finally we summarize and conclude our work in Section 7.

2. TECHNICAL DESCRIPTION
Dynamic taint tracking is the process of marking (tainting) and following (tracking) certain pieces of data as they are used during the execution of a program. The process can be subdivided into three core tasks: tainting desired values at their sources, propagating taint when tainted values are used and enforcing policies which ensure that tainted values are not used in pre-defined ways during execution.

2.1 Source Tainting
For most applications of dynamic taint tracking for web browsing systems, tainted values need to originate from user-provided data and from certain properties of DOM objects such as document.URL, document.referrer, document.location, and window.location. It is important to taint these sources before they are used because they hold information that can be abused by an adversary to launch an attack to surreptitiously gain sensitive user information. Table 1 summarizes a list of sources which should be initially tainted for testing an implementation of dynamic taint tracking for web browsing systems.

2.2 Taint Propagation
In any taint tracking system, taint must be propagated - copied from one tainted value to another - according to a set of rules. For dynamic taint tracking in web browsing systems, these rules are best described by the following set of semantics:

- Literals (e.g. true, false, 1, 2, -1.0, quoted string) are never tainted.
- The resulting value of any evaluated expression (e.g. +, -, in, return) whose result depends on a tainted value is tainted.
- Only the data-holding properties of a JavaScript object can be tainted, not the object itself.
- Tainting one property of an object only taints the data referred to by that property and not other properties

\(^1\)The Document Object Model (DOM) is a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents displayed in a Web browser.
of the object (unless both properties are aliased to the same data).
- Since functions are properties of objects, they may become tainted. The return value of a tainted function is always tainted, even for functions with implicit return statements (e.g. constructors).
- Taint does not propagate between values in different scopes solely due to variable shadowing.

### 2.3 Tracking Policies

As taint propagates during the execution of a document’s associated JavaScript, actions need to be taken to prevent the use of tainted values in certain situations. An example of such a situation would be the execution of the `eval` function on a tainted value. Collectively these situations can be viewed as a set of security policies for protecting user-sensitive data as well as internal application state. We suggest prohibiting the execution of any tainted data or the assignment of DOM object properties described in Table 1 to tainted values for protection against a wide range of domain-crossing and SQL-injecting exploits.

### 2.4 TaintSNIFFER Implementation

We extend the C3 browser’s Javascript engine, SPUR [1], and its DOM subsystem to dynamically taint and track user-provided data and sensitive internal data of certain DOM nodes. Like other JavaScript engines, SPUR initially interprets executing JavaScript code at a bytecode level to trace execution paths for future optimization. We instrument SPUR’s bytecode interpreter to implement our dynamic taint tracking system. We also add taint fields to C3’s internal representation of DOM nodes so that we can taint their properties as necessary.

#### 2.4.1 Taint Representation

We taint values by adding and setting a boolean flag of each JavaScript object property which contains the data. While this approach incurs a minimum one byte of overhead for each object property, constant time taint-checking and taint propagation are preserved. Additionally, in the C3 system JavaScript values are already boxed with a generic wrapper object.

#### 2.4.2 Source Tainting

We add two functions, `taint()` and `untaint()`, to SPUR’s set of pre-defined functions for programmatically tainting and clearing the taint of JavaScript variables. This allows us to taint the property `p` of an object `o` with the statement `o.p = taint(o.p)` and clear its taint with the statement `o.p = untaint(o.p)`. This idea can be extended for tainting the initial properties of the global DOM `window`, `document` and `history` objects in the `window.onload` function which is invoked immediately after the browser completes parsing the associated document’s XHTML content into its DOM representation.

#### 2.4.3 Taint Propagation

We propagate taint according to the set of rules outlined in Section 2.2.

#### 2.4.4 Tracking Policies

We add an additional function, `checkTaint()`, to SPUR’s set of pre-defined functions for checking the taint of a variable at any point in execution. `checkTaint(x)` returns `true` if `x` is tainted and `false` otherwise. We call `checkTaint()` at various points in our test suite for verifying the correct functioning of our dynamic taint tracking system.

### 3. RELATED WORK

The Netscape group was one of the first to employ dynamic taint tracking within a browser’s JavaScript engine for protecting against domain-crossing exploits [2]. The group outlined what sources must be tainted for effective prevention and proposed including a domain name label for each tainted variable to track its origin. We adopt their set of must-taint sources, listed in bold in Table 1 and several of their suggested sources for initial tainting, also summarized in the same table.

Recently several groups have implemented their own Web browsers which use dynamic taint tracking to prevent against known domain-crossing attacks and exploits. Tang et al.’s Alhambra browser [3] uses fine-grained security policies to increase the robustness of policy-enhanced web browsing via a dynamic taint-tracking system built into the JavaScript and DOM subsystems. We adopt their rules for propagating taint at the JavaScript object level when tainted object properties are used in an assignment, logical, arithmetic or string manipulating operation.

Other recent work has extended existing Web browsers to incorporate and evaluate the use of dynamic taint tracking for exploit and attack prevention. Vogt et al. extend Mozilla’s Firefox browser to perform both static and dynamic taint tracking [4]. Their static component is used to cover information flows which cannot be dynamically detected which attackers are free to use to launch domain-crossing attacks. An example of such a flow is the use of a tainted value in

<table>
<thead>
<tr>
<th>Object</th>
<th>Tainted properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>cookie, domain, forms, lastModified, links, referer, title, URL</td>
</tr>
<tr>
<td>Form</td>
<td>action</td>
</tr>
<tr>
<td>Any form input element</td>
<td>checked, defaultChecked, defaultValue, name, selectedIndex, toString, value</td>
</tr>
<tr>
<td>History</td>
<td>current, next, previous, toString</td>
</tr>
<tr>
<td>Select option</td>
<td>defaultSelected, selected, text, value</td>
</tr>
<tr>
<td>Location and Link</td>
<td>hash, host, hostname, href, pathname, port, protocol, search, toString</td>
</tr>
<tr>
<td>Window</td>
<td>defaultStatus, status, location</td>
</tr>
</tbody>
</table>

Table 1: Sensitive Data Sources. We initially focus on those in bold.
a logical operation where the result short-circuits around a tainted operand. We adopt their rules for propagating taint to the return value of function calls, including eval() requests.

Some recent work focuses on protecting both the client and server side resources used during the lifetime of a web application. Xu et. al. use fine-grained taint analysis in [5] to strengthen security in the browser and in server-side scripts and applications by augmenting security policies with information about the trustworthiness of data used in security-sensitive operations. Taints are represented as a boolean array called the tagmap which is indexed with a variable’s memory address and propagated in a similar fashion as our work. By using a tagmap for tainting and tracking the clientside execution of a web page’s JavaScript as well as a tagmap for taint tracking during the execution of any server-side scripts they are able to effectively prevent against a wide range of web-based attacks including SQL- and command-injection, cross-site scripting, format string corruption, memory error exploits and access-privilege-hijacking attacks. We leverage their idea of adding functions for tainting, untainting, and checking the taint of a value via its container to our JavaScript engine for evaluating the correctness of our taint propagation rules and performing intermediate, unit-style testing during development.

None of the aforementioned related works discuss dynamic taint tracking issues related to scoping and object prototypes. Hence, we focus a significant portion of our test suite development to creating tests which propagate taint through object prototype properties. We also test issues related to variable shadowing and lexical environment closures for ensuring taint is propagated with respect to scope.

4. EXPERIMENTAL METHODOLOGY

We evaluate TaintSNIFFER along lines of semantic correctness. We add a basic test framework written in C# to C3’s JavaScript engine for our evaluation. This framework exposes a built-in function called assertFunc that accepts four arguments: the result of invoking checkTaint() on a JavaScript variable, the expected result, a test class description, and a detailed test description. We write test cases in JavaScript and report their results by calling the assertFunc function. The framework executes all the JavaScript files in a specified directory of test cases which use the assertFunc function, executes each case with our modified version of SPUR’s bytecode interpreter and outputs the results of all the test cases in the specified directory with additional information about any cases which failed.

Overall the semantic correctness of our taint tracking sub-system is separated into five categories: core operations, simple operations, object operations, scope operations, and DOM operations.

4.1 Core Operations

The core operations tests verify the correct functioning of our taint(), untaint() and checkTaint() functions. The tests not only verify that a variable has been correctly tainted but also that no other objects or properties are tainted because of the previous taint operation. This is necessary since a variable in a JavaScript function is actually a property of the global or the enclosing scope’s object and an incorrectly implemented taint() function can pollute the global object and all of its properties. Also, constant literals are tested to make sure that they are never tainted (yet when they are used in an expression, the taint is propagated). Refer to Appendix Section A.1 for more details.

4.2 Simple Operations

The simple operational tests verify that taint is propagated when tainted values are used as operands in arithmetic, logical, bitwise, comparison, string and complex assignment expressions. They exhaustively test that the taint is propagated whenever a tainted value is used in an expression consisting of the aforementioned “simple” operators. Refer to Appendix Section A.2 for more details.

4.3 Object Operations

The object operational test cases cover different operations which modify object properties and object prototype properties. These are important to test since JavaScript is a loosely-typed prototype based language. It is natural to have mutable data in the objects and any common (immutable) properties as a part of an object’s prototype property (which is shared by every object instance constructed from the same constructor). Hence, it can be disastrous to accidentally taint the prototype property of an object since then every object which has a reference to the same prototype are tainted. In general, these tests add properties and functions to the prototype dynamically and test for the correct behavior, initialize an object from tainted data, and also ensure that not every object becomes tainted.

For example, Figure 1 illustrates a block of code from Appendix Section A.3 that illustrates one of our object operations test cases. In Figure 1 the last two checkTaint() calls are worth noticing, since hello() is a function that returns the name property, which is tainted for the object myTaintedPropertyPet and not tainted for myPet. Refer to Appendix Section A.3 for more details.

4.4 Scope Operations

JavaScript introduces a new scope only in the context of a function call. Variables declared anywhere inside a function are semantically equivalent to the variables declared in the beginning (except nested functions). Moreover, the JavaScript uses lexical scoping rules for functions, which implies that the scope chain inside a function is always the same. This leads to interesting behavior such as closures when used in combination with nested functions. It is important to make sure that any variables in a closure retain their taint across different execution contexts. Another important test case is when scope changes with the use of the with operator as the shadowed properties have their taint status preserved as well as the other properties with the same name that are now in the current scope have their taint status preserved.

For example, Figure 2 illustrates one of the test cases where the InsideScope function toggles storing its property from tainted to untainted. Refer to Appendix Section A.4 for more details.
// Simple Object
var Pet = function (name, gender) {
    if (!this instanceof Pet) {
        return new Pet(name, gender);
    }
    this.name = name;
    this.gender = gender;
    this.hello = function () {
        return "Hello, I am " + name + ".";
    }
}
var myPet = new Pet("T", 'M');
var taintedName = "J";
taintedName = taint(taintedName);
var myTaintedPropertyPet = new Pet(taintedName, 'F');
var myTPPHello = myTaintedPropertyPet.hello();

// Should be true
checkTaint(myTaintedPropertyPet.hello());
// Should be false
checkTaint(myPet.hello());

Figure 1: An example test case for the Object Operation Test case category.

function Scoping(a, b) {
    var exposedProperty = "";
    var taintedClosure = taint("taint");
    var untaintedClosure = "untaint";
    var boolValue = false;
    function InsideScope() {
        if (boolValue) {
            exposedProperty = taintedClosure;
        } else {
            exposedProperty = untaintedClosure;
        }
        boolValue = !boolValue;
        return exposedProperty;
    }
    return InsideScope;
}
var closedScope = Scoping(2, 3);
checkTaint(closedScope()); // Should be false
checkTaint(closedScope()); // Should be true
checkTaint(closedScope()); // Should be false
checkTaint(closedScope()); // Should be true

Figure 2: An example test case for the Scope Operation Test case category.

window.onload = function() {
    var someStr = "Hello World!";
    someStr = taint(someStr);
    alert(checkTaint(someStr));
    // Append the tainted string into some DOM element
    var p = document.getElementById('a_p');
p.textContent += someStr;
    // textContent should be tainted
    alert("node('a_p').textContent isTainted = " + checkTaint(p.textContent));
    // innerHTML should NOT be tainted
    alert("node('a_p').innerHTML isTainted = " + checkTaint(p.innerHTML));
    // p should be tainted
    alert("node('a_p') isTainted = " + checkTaint(p));
    // window (global obj) should not be tainted
    alert("window isTainted = " + checkTaint(window));
}

Figure 3: An example test case for the Scope Operation Test case category.

4.5 DOM Operations
The DOM operations test cases examine the storage of tainted values into properties of DOM objects and the use of tainted values stored in DOM object properties to verify that taint is preserved. The code shown in Figure 3 illustrates a very simple example where a tainted string is stored in a property of a paragraph element and is later retrieved. Refer to Appendix Section A.5 for more details.

5. RESULTS AND DISCUSSION
5.1 Core Operations
The results of our core operations tests (summarized in Table 2 of Appendix B) demonstrate the correct functioning of our \textit{taint()} and \textit{untaint()} additions to the set of pre-defined functions of C3’s JavaScript engine. The results of the tests also show the correct propagation of taint through direct assignment and the clearing of taint by assignment of a previously tainted variable to an untainted value.

5.2 Simple Operations
The results from our simple operations tests (summarized in Table 3 of Appendix B) demonstrate the correct propagation of taint for arithmetic, logical, bitwise, comparison, and compound assignment expressions in which one or more source values that the resulting value depends upon is tainted. It is important to note, as is the case with the logical AND (\&\&) and logical OR (||) operations, that taint is not propagated to the result when the logic “short circuits” around a tainted operand. This behavior highlights the dynamic nature of our system.

5.3 Object Operations
The results of our object-based testing regarding object properties, prototypes, and built-in JavaScript objects (e.g. Ar-
ray, Date) demonstrate our strict adherence to keeping taint information as close to the data which it is associated with as possible. We achieve this goal by tainting the properties of an object, rather than an object itself (since they are what hold the data). For functional object properties, our implementation taints the entire function and therefore its return value as outlined in our taint propagation semantics in Section 2.2. We only fail one test case in this test category, the case where the contents of an Array which contains tainted values are joined together. This shortcoming is due to our system’s inability to propagate taint from within a function which returns a tainted value. The full results of these tests are summarized in Table 4 of Appendix B.

5.4 Scope Operations
The results from our scoping tests (summarized in Table 5 of Appendix B) raise several issues with how scope is handled by our system. Firstly, our false reporting of newObj.a’s property in the scope of with(neobj)) indicates that the proper scope is not being explored upon checking the taint of newObj.a. This may be a shortcoming of our implementation, or may be a failure in the current state of the C3 JavaScript engine in handling such a scope. We are unable to determine which assessment is correct without further research. Secondly, our false reporting of closedScope()’s return value in our elementary scoping test highlights a deeper issue related to returning tainted values from a function or any nested scope. Finally, our test case failures in the if-else conditional case shows our lack of implementation for such cases where information leakage about tainted data is possible.

5.5 DOM Operations
Our test cases are not fully exhaustive for testing the manipulation of DOM objects by JavaScript. However, they suffice for confirming the inability to launder taint through the DOM and therefore only focus on ensuring that any tainted values stored into the DOM remain tainted until they are reassigned to untainted values. Our system passes both the case of writing tainted data into the property of a DOM object and reading tainted data from a property of a DOM object by JavaScript code. The full results for these tests are summarized in Table 6 of Appendix B.

6. LIMITATIONS
There are several limitations with our test suite and dynamic taint tracking system. Most notably, the C3 browser’s Document Object Model is not yet fully implemented and still lacks components of essential nodes (input, textarea, form) for collecting user input from an XHTML-formatted documents. Furthermore, the cases in our test suite may not be fully exhaustive and certainly require further study and experimentation. Since our ideal application of TaintSNIFFER falls within the realm of web security, there may exist loopholes unknown to us in the C3 browser and its subsystems which require patching. Finally, TaintSNIFFER’s requirement that all JavaScript be interpreted by SPUR significantly hampers the performance gains that SPUR affords with its tracing just-in-time compilation opportunities.

In this project we developed TaintSNIFFER, a dynamic taint tracking system within Microsoft Research’s novel homogenous C3 web browser. By taking a test-driven approach for implementing TaintSNIFFER we were able to compile a robust and practical test suite for use in the development of dynamic taint tracking systems for JavaScript and web browsers in general. We began our project by studying previous and current uses of dynamic taint tracking in web browsing systems for preventing the clandestine transfer of sensitive information by domain-crossing exploits. Upon reviewing this work, we noticed a lack of information regarding taint propagation in the properties and prototypes of JavaScript objects. It was at this point that we shelved our plans for extending and evaluating TaintSNIFFER’s ability to prevent the transfer of sensitive information by domain-crossing exploits and redirected our focus to authoring a rich suite of tests for dynamic taint tracking systems in the JavaScript language.

By categorizing our tests into five distinct categories we were able to rapidly author unit-style tests which we then validated with TaintSNIFFER. While TaintSNIFFER did not correctly propagate taint in all of the test cases, we were able to identify concrete shortcomings in our implementation. We also were able to extend our test suite to include the DOM subsystem and experienced success in preventing the laundering of tainted values through the DOM. Given the current state of TaintSNIFFER’s implementation and the coverage of our test suite we expect handling the failing cases to be a simple process involving further study of the C3 JavaScript engine’s bytecode interpreter. Once we verify TaintSNIFFER’s robustness, we can move forward with adding a policy engine for preventing the compromise of sensitive data by domain-crossing attacks.

8. REFERENCES
APPENDIX
A. TEST SUITE
A.1 Core Operations

// Basic taint functionality
//
var a = taint(a);
assertFunc(checkTaint(a), true, "Taint", "Simple variable tainted");
assertFunc(checkTaint(this.a), true, "Global", "Property of global object tainted");
assertFunc(checkTaint(a), false, "Global", "Global object untainted");

var b = a;
assertFunc(checkTaint(b), true, "Assignment", "Simple assignment statement");
assertFunc(checkTaint(this.b), true, "Global", "Property of global object tainted");
assertFunc(checkTaint(b), false, "Global", "Global object untainted after assignment statement");

a = untaint(a);
assertFunc(checkTaint(a), false, "Untaint", "Simple variable untainted");
assertFunc(checkTaint(this.a), false, "Global", "Property of global object untainted");
assertFunc(checkTaint(a), true, "Persist", "Taint from assignment persists");
assertFunc(checkTaint(this.b), true, "Global", "Property assigned to of global object remains tainted");
assertFunc(checkTaint(this), false, "Global", "Global object remains untainted");

// Reassinging...
//
b = "untainted";
assertFunc(checkTaint(b), false, "Reassign", "Untaint due to reassignment");

// Tainting Literals
//
var a1 = taint(true);
var a2 = taint("Hello World");
var a3 = taint(1);
var a4 = taint(-1.0);
assertFunc(checkTaint(a1), true, "taint literals", "boolean assigned variable");
assertFunc(checkTaint(a2), true, "taint literals", "string assigned variable");
assertFunc(checkTaint("Hello World"), false, "taint literals", "string literal");
assertFunc(checkTaint(a3), true, "taint literals", "integers/floats assigned variable");
assertFunc(checkTaint(1), false, "taint literals", "integers/floats literal");
assertFunc(checkTaint(-1.0), false, "taint literals", "floats literal");

A.2 Simple Operations

// Arithmetic
var a = 10;
a = taint(a);

var b = a + 4;
var b1 = 4 + a;
assertFunc(checkTaint(b), true, "Binary Add", "b = a + 4");
assertFunc(checkTaint(b1), true, "Binary Add", "b1 = 4 + a");

var c = a - 4;
var c1 = 4 - a;
assertFunc(checkTaint(c), true, "Binary Subtract", "c = a - 4");
assertFunc(checkTaint(c1), true, "Binary Subtract", "c1 = 4 - a");

var d = a * 2;
var d1 = 2 * a;
assertFunc(checkTaint(d), true, "Binary Multiply", "d = a * 2");
assertFunc(checkTaint(d1), true, "Binary Multiply", "d1 = 2 * a");

var e = a / 2;
var e1 = 2 / a;
assertFunc(checkTaint(e), true, "Binary Division", "e = a / 2");
assertFunc(checkTaint(e1), true, "Binary Division", "e1 = 2 / a");

var f = a % 4;
var f1 = 4 % a;
assertFunc(checkTaint(f), true, "Binary Modulus", "f = a % 4");
assertFunc(checkTaint(f1), true, "Binary Modulus", "f1 = 4 % a");

var g = a;
assertFunc(checkTaint(g), true, "Unary Plus", "g = +a");

var h = -a;
assertFunc(checkTaint(h), true, "Unary Minus", "h = -a");

a++;
++a;
assertFunc(checkTaint(a), true, "Postfix increment", "a++");
assertFunc(checkTaint(a), true, "Prefix increment", "++a");
a--;
--a;
assertFunc(checkTaint(a), true, "Postfix decrement", "a--");
assertFunc(checkTaint(a), true, "Prefix decrement", "-a");
/
var k = 4;
k += a;
assertFunc(checkTaint(k), true, "Composite increment", "k += a");

var l = 6;
l -= a;
assertFunc(checkTaint(l), true, "Composite decrement", "l -= a");
*/

// Bitwise
var aa = a & 4;
var aa1 = 4 & a;
assertFunc(checkTaint(aa), true, "Bitwise And", "aa = a & 4");
assertFunc(checkTaint(aa1), true, "Bitwise And", "aa1 = 4 & a");

var bb = a | 5;
var bb1 = 5 | a;
assertFunc(checkTaint(bb), true, "Bitshift Left", "bb = a | 5");
assertFunc(checkTaint(bb1), true, "Bitshift Left", "bb1 = 5 | a");

var cc = "a;";
assertFunc(checkTaint(cc), true, "Bitwise Not", "cc = "a;"");

var dd = a << 3;
var dd1 = 3 << a;
assertFunc(checkTaint(dd), true, "Bitshift Left", "dd = a << 3");
assertFunc(checkTaint(dd1), true, "Bitshift Left", "dd1 = 3 << a");

var ee = a >> 1;
var ee1 = 204050 >> a;
assertFunc(checkTaint(ee), true, "Bitshift Right", "ee = a >> 1");
assertFunc(checkTaint(ee1), true, "Bitshift Right", "ee1 = 204050 >> a");

var ff = a >>> 1;
var ff1 = 204050 >>> a;
assertFunc(checkTaint(ff), true, "Shift Right With Sign", "ff = a >>> 1");
assertFunc(checkTaint(ff1), true, "Shift Right With Sign", "ff1 = 204050 >>> a");

var gg = a ^ 4;
var gg1 = 4 ^ a;
assertFunc(checkTaint(gg), true, "XOR", "gg = a ^ 4");
assertFunc(checkTaint(gg1), true, "XOR", "gg1 = 4 ^ a");

// Logical
var a1 = true;
//a1 = taint(a1);
var a2 = true;
a2 = taint(a2);

var aaa = a1 && a2;
var aa1 = a2 && a1;
var aa2 = true && a2;
var aa3 = a2 && false;
assertFunc(checkTaint(aaa), true, "Logical And", "aaa = a1 && a2");
assertFunc(checkTaint(aa1), false, "Logical And", "aa1 = a2 && a1");
assertFunc(checkTaint(aa2), true, "Logical And", "aa2 = true && a2");
assertFunc(checkTaint(aa3), false, "Logical And", "aa3 = a2 && false");

var bbb = a2 || a1;
var bbb1 = a1 || a2;
var bbb2 = a2 || false;
var bbb3 = true || a2;
assertFunc(checkTaint(bbb), true, "Logical Or", "bbb = a2 || a1");
assertFunc(checkTaint(bbb1), false, "Logical Or", "bbb1 = a1 || a2");
assertFunc(checkTaint(bbb2), true, "Logical Or", "bbb2 = a2 || false");
assertFunc(checkTaint(bbb3), false, "Logical Or", "bbb3 = true || a2");

var ccc = !a2;
assertFunc(checkTaint(ccc), true, "Logical Not", "ccc = !a2");

// Comparison
var aaaa = a < 2;
var aaaa1 = 2 < a;
assertFunc(checkTaint(aaaa), true, "Less Than", "aaaa = a < 2");
assertFunc(checkTaint(aaaa1), true, "Less Than", "aaaa1 = 2 < a");

var bbbb = a > 4;
var bbbb1 = 4 > a;
assertFunc(checkTaint(bbbb), true, "Greater Than", "bbbb = a > 4");
assertFunc(checkTaint(bbbb1), true, "Greater Than", "bbbb1 = 4 > a");

var cccc = a <= 2;
var cccc1 = 2 <= a;
assertFunc(checkTaint(cccc), true, "Less Than Or Equal", "cccc = a <= 2");
assertFunc(checkTaint(cccc1), true, "Less Than Or Equal", "cccc1 = 2 <= a");

var dddd = a >= 4;
var dddd1 = 4 >= a;
assertFunc(checkTaint(dddd), true, "Greater Than Or Equal", "dddd = a >= 4");
assertFunc(checkTaint(dddd1), true, "Greater Than Or Equal", "dddd1 = 4 >= a");

var eeee = a == 2;
var eeee1 = 2 == a;
assertFunc(checkTaint(eeee), true, "Equals Equals", "eee = a == 2");
assertFunc(checkTaint(eeee1), true, "Equals Equals", "eee1 = 2 == a");

var ffff = a != 2;
var ffff1 = 2 != a;
assertFunc(checkTaint(ffff), true, "Not Equals", "ffff = a != 2");
assertFunc(checkTaint(ffff1), true, "Not Equals", "ffff1 = 2 != a");

// String
var a3 = "asdf";
a3 = taint(a3);
var aaaaa = a3 + 2;
var aaaaal = 2 + a3;
assertFunc(checkTaint(aaaaa), true, "String Concat.", "aaaaa = a3 + 2");
assertFunc(checkTaint(aaaaal), true, "String Concat.", "aaaaa = 2 + a3");
var bbbbbb = a3 < "foo";
var bbbbl = "foo" < a3;
assertFunc(checkTaint(bbbbbb), true, "String Less Than", "bbbbl = a3 < "foo"");
assertFunc(checkTaint(bbbbl), true, "String Less Than", "bbbbl = "foo" < a3");
var ccccc = a3 > "foo";
var ccccl = "foo" > a3;
assertFunc(checkTaint(cccccc), true, "String Greater Than", "ccccc = a3 > "foo"");
assertFunc(checkTaint(cccccl), true, "String Greater Than", "cccccl = "foo" > a3");
var ddddd = a3 <= "foo";
var ddddl = "foo" <= a3;
assertFunc(checkTaint(dddddd), true, "String Less Than Or Equal", "ddddd = a3 <= "foo"");
assertFunc(checkTaint(dddddl), true, "String Less Than Or Equal", "ddddl = "foo" <= a3");
var eeeeee = a3 >= "foo";
var eeeel = "foo" >= a3;
assertFunc(checkTaint(eeeeee), true, "String Greater Than Or Equal", "eeee = a3 >= "foo"");
assertFunc(checkTaint(eeeel), true, "String Greater Than Or Equal", "eeel = "foo" >= a3");

// Complex Assignment
var aaaaa = 4;
var aaaaal = 2 + a3;
assertFunc(checkTaint(aaaaa), true, "Plus Equals", "aaaaa += a");
assertFunc(checkTaint(aaaaal), true, "Plus Equals", "aaaaal += a");
var bbbbbb = 4;
var bbbbl = a3;
assertFunc(checkTaint(bbbbbb), true, "Minus Equals", "bbbbl -= a");
assertFunc(checkTaint(bbbbl), true, "Minus Equals", "bbbl -= a");
var ccccc = 4;
var ccccl = a3;
assertFunc(checkTaint(cccccc), true, "Times Equals", "ccccc *= a");
assertFunc(checkTaint(cccccl), true, "Times Equals", "cccl *= a");
var ddddd = 0;
var ddddl = a3;
assertFunc(checkTaint(dddddd), true, "Divide Equals", "ddddd /= a");
assertFunc(checkTaint(dddddl), true, "Divide Equals", "dddl /= a");
var eeeeee = 3;
var eeeel = a3;
assertFunc(checkTaint(eeeeee), true, "Modulo Equals", "eeee = a3 % a");
assertFunc(checkTaint(eeeel), true, "Modulo Equals", "eeel = a3 % a");
var fffffff = 2;
var ffffl = a3;
assertFunc(checkTaint(ffffff), true, "Shift Left Equals", "ffffff <<= a");
assertFunc(checkTaint(ffffff), true, "Shift Left Equals", "ffffff <<= a");
var gggggg = 2048;
var ggggg = 2048;
assertFunc(checkTaint(gggggg), true, "Shift Right Equals", "ggggg >> = a");

A.3 Object Operations

// Simple Object
var Pet = function (name, gender) {
    if (!this instanceof Pet) {
        return new Pet(name, gender);
    }
    this.name = name;
    this.gender = gender;
    this.hello = function () { return "Hello, I am " + name + ", ";
    }
}

var myPet = new Pet("Trevor", "M");
var myTaintedPropertyPet = new Pet(taintedName, "M");
assertFunc(checkTaint(myPet), false, "Simple Object", "Untainted construction of simple object");
assertFunc(checkTaint(myTaintedPropertyPet), false, "Simple Object", "Untainted prototype of untainted simple object");

var taintedName = "Justine";
taintedName = taint(taintedName);
var myTaintedPropertyPet = new Pet(taintedName, "F");
var myTPPHello = myTaintedPropertyPet.hello();
assertFunc(checkTaint(myTaintedPropertyPet.name), true, "Simple Object", "Tainted object property");
assertFunc(checkTaint(myTaintedPropertyPet.hello), false, "Simple Object", "Untainted property of object with tainted property");
assertFunc(checkTaint(myTPPHello), true, "Simple Object", "Tainted return value of object property which uses tainted property");

assertFunc(checkTaint(myTaintedPropertyPet), false, "Simple Object", "Untainted object with tainted property");
assertFunc(checkTaint(myTaintedPropertyPet.name), false, "Simple Object", "Untainted name property of untainted Object Pet");

var myTaintedPet = new Pet("Param", "M");
myTaintedPet = taint(myTaintedPet);
assertFunc(checkTaint(myTaintedPet), true, "Simple Object", "Tainted object");
assertFunc(checkTaint(myTaintedPet.name), false, "Simple Object", "Property of tainted object");
assertFunc(checkTaint(myTaintedPet.prototype),
false, "Simple Object", "Untainted prototype of tainted object");

// Prototypes
var x = "Aaron";
x = taint(x);
Pet.prototype.owner = x;

var newPet = new Pet("Spike", 'M');
assertFunc(checkTaint(newPet.owner), true, "Prototype", "Tainted prototype property");
assertFunc(checkTaint(myPet.owner), true, "Prototype", "Shared tainted prototype property");
Pet.prototype = taint(Pet.prototype);
assertFunc(checkTaint(newPet), false, "Prototype", "Object instance w/ tainted prototype untainted");
assertFunc(checkTaint(newPet.prototype), false, "Prototype", "Tainted prototype property of object");

Pet.prototype.greet = function () { return "Hi, I'm " + this.name + " and I belong to " + this.owner;);
var g = newPet.greet();
assertFunc(checkTaint(g), true, "Prototype", "Return value of prototype method which uses tainted data");

// Array
var arrElement = "foo";
arrElement = taint(arrElement);

var simpleArr = new Array(1, 2, "duck", arrElement, "orange");
assertFunc(checkTaint(simpleArr), false, "Array", "Array after initialization w/ tainted value");
assertFunc(checkTaint(simpleArr[3]), true, "Array", "Array access of tainted value");

var isInArr = arrElement in simpleArr;
assertFunc(checkTaint(isInArr), true, "In", "Result of in operation looking for tainted value");

var joinedArrContents = simpleArr.join();
assertFunc(checkTaint(joinedArrContents), true, "Array", "Joined array contents containing tainted data");

var concatArr = simpleArr.concat("stuff");
assertFunc(checkTaint(concatArr), false, "Array", "Result of array concatenation");
assertFunc(checkTaint(concatArr[3]), true, "Array", "Element in result of array concatenation");

var sliceArr = simpleArr.slice(1, -1);
assertFunc(checkTaint(sliceArr), false, "Array", "Result of array slice");
assertFunc(checkTaint(sliceArr[2]), true, "Array", "Element in result of array slice");

var assocArr = new Array();
assocArr[arrElement] = "Hello World!";
assertFunc(checkTaint(assocArr[arrElement]), false, "Array", "Associative array access w/ tainted key");
// debugger;
assocArr["asdf"] = arrElement; // same as assocArr.asdf = arrElement;
assertFunc(checkTaint(assocArr["asdf"]), true, "Array", "Associative array access of tainted value"); // Fails
assertFunc(checkTaint(assocArr), false, "Array", "Associative array with tainted key and tainted value");

A.4 Scope Operations

// with tests

var newObj = { a: "foo", b: "blah" };
newObj.a = taint(newObj.a);
var a = "untainted";
assertFunc(checkTaint(a), false, "With test", "'a' untainted in global scope");
assertFunc(checkTaint(newObj.a), true, "With test", "'a' untainted in global scope");
assertFunc(checkTaint(newObj), true, "With test", "obj.a tainted in global scope");

with (newObj) {
    print("Value of a is: " + a);
    assertFunc(checkTaint(a), true, "With test", "'a' untainted in newObj scope");
    var c = a;
    assertFunc(checkTaint(c), true, "With test", "One of the vars assigned from the tainted obj property");
    var d = b;
    assertFunc(checkTaint(d), false, "With test", "One of the vars assigned from the untainted obj property");
}
assertFunc(checkTaint(a), false, "With test", "a remains untainted in the global scope");

// elementary closure test

function Scoping(a, b) {
    var exposedProperty = "";
    var taintedClosure = "taint";
    var untaintedClosure = "untaint";
    var boolValue = false;

    function InsideScope() {
        if (boolValue) {
            exposedProperty = taintedClosure;
        } else {
            exposedProperty = untaintedClosure;
        }
        print("Boolean is: " + boolValue);
        print("Taint is: " + checkTaint(exposedProperty));
    }
boolValue = !boolValue;

return taintedClosure;
}

return InsideScope;
}

var closedScope = Scoping(2, 3);
assertFunc(checkTaint(closedScope()), false, "Closure Test", "First time return value is untainted");
assertFunc(checkTaint(closedScope()), true, "Closure Test", "Next time return value IS tainted");
assertFunc(checkTaint(closedScope()), false, "Closure Test", "Third time return value is again untainted");
assertFunc(checkTaint(closedScope()), true, "Closure Test", "Next time return value IS tainted (and the cycle repeats)");

// elementary closure test: Setting/Getting of private object properties.

function makeProperty(o, name, predicate) {
    var value; // This is the property value
    // The getter method simply returns the value.
    o["get" + name] = function () { return value; }
    // The setter method stores the value or throws an exception if
    // the predicate rejects the value.
    o["set" + name] = function (v) {
        if (predicate && !predicate(v))
            throw "set" + name + ": invalid value 
        else
            value = v;
    }
}

// The following code demonstrates the makeProperty() method.
var o = {}; // Here is an empty object
// Add property accessor methods getName and setName()
// Ensure that only string values are allowed
makeProperty(o, "Name", function (x) { return typeof x == "string"; });
o.setName("Frank"); // Set the property value
print(o.getName()); // Get the property value
var taintedName = "Tainted Name"
taintedName = taint(taintedName);
assertFunc(checkTaint(o.getName()), false, "Private Scoped Props", "Setting prop to untainted value");
o.setName(taintedName);
assertFunc(checkTaint(o.getName()), true, "Private Scoped Props", "Setting prop name to tainted value");
o.setName("Frank");
assertFunc(checkTaint(o.getName()), false, "Private Scoped Props", "Setting prop name back to untainted value");

// conditionals

function TestTaintReturn() {
    var a = taint("taint");
    return a;
}
//debugger;
var abc = checkTaint(TestTaintReturn());
assertFunc(abc, true, "Function tainted return", "Function returns a tainted value");

// conditionals

var a = "if taint test"
var b = true;
b = taint(b);
if (a && b) {
    var c = a;
} else (b)
{
    var c = b;
A.5 DOM Operations

```html
<html>
<head>
  <script type="text/javascript">
    window.onload = function() {
      var someStr = "Hello World!";
      someStr = taint(someStr);
      var p = document.getElementById('a_p');
      p.textContent += someStr;
      assertFunc(checkTaint(p.textContent), true, "DOM Write", "node's textContent property tainted");
      assertFunc(checkTaint(p.innerHTML), true, "DOM Write", "node's innerHTML property tainted");
      assertFunc(checkTaint(p), false, "DOM Write", "node tainted");
      var someOtherStr = document.getElementById('a_p').textContent;
      assertFunc(checkTaint(someOtherStr), true, "DOM Read", "assigning to node's textContent propagates taint");
      var someUntaintedOtherStr = document.getElementById('other_p').textContent;
      assertFunc(checkTaint(someUntaintedOtherStr), false, "DOM Read", "other node's textContent property untainted");
    }
  </script>
</head>
<body>
  <p id="a_p">This is a paragraph.</p>
</body>
</html>
```
<table>
<thead>
<tr>
<th>Test Case</th>
<th>Description</th>
<th>Passed</th>
<th>Expected</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taint</td>
<td>Simple variable tainted</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Taint</td>
<td>Simple variable tainted</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Global</td>
<td>Property of global object tainted</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Global</td>
<td>Global object untainted</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Assignment</td>
<td>Simple assignment statement</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Global</td>
<td>Property of global object tainted</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Global</td>
<td>Global object untainted after assignment statement</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Untaint</td>
<td>Simple variable untainted</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Global</td>
<td>Property of global object untainted</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Global</td>
<td>Global object untainted</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Persist</td>
<td>Taint from assignment persists</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Global</td>
<td>Property assigned to of global object remains tainted</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Global</td>
<td>Global object remains untainted</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Reassign</td>
<td>Untaint due to reassignment</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>taint literals</td>
<td>boolean assigned variable</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>taint literals</td>
<td>boolean literal</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>taint literals</td>
<td>string assigned variable</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>taint literals</td>
<td>string literal</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>taint literals</td>
<td>integers/floats assigned variable</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>taint literals</td>
<td>integers/floats literal</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>taint literals</td>
<td>floats literal</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Table 2: Core Operations Test Results
<table>
<thead>
<tr>
<th>Test Case</th>
<th>Description</th>
<th>Passed</th>
<th>Expected</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Add</td>
<td>$b = a + 4$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Binary Add</td>
<td>$b1 = 4 + a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Binary Subtract</td>
<td>$c = a - 4$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Binary Subtract</td>
<td>$c1 = 4 - a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Binary Multiply</td>
<td>$d = a * 2$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Binary Multiply</td>
<td>$d1 = 2 * a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Binary Division</td>
<td>$e = a / 2$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Binary Division</td>
<td>$e1 = 2 / a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Binary Modulus</td>
<td>$f = a % 4$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Binary Modulus</td>
<td>$f1 = 4 % a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Unary Plus</td>
<td>$g = +a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Unary Minus</td>
<td>$h = -a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Postfix increment</td>
<td>$a++$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Prefix increment</td>
<td>$++a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Postfix decrement</td>
<td>$a--$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Prefix decrement</td>
<td>$--a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Bitwise And</td>
<td>$aa = a &amp; 4$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Bitwise And</td>
<td>$aa1 = 4 &amp; a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Bitwise Or</td>
<td>$bb = a</td>
<td>5$</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Bitwise Or</td>
<td>$bb1 = 5</td>
<td>a$</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Bitwise Not</td>
<td>$cc = a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Bitshift Left</td>
<td>$dd = a &lt;&lt; 3$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Bitshift Left</td>
<td>$dd1 = 3 &lt;&lt; a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Bitshift Right</td>
<td>$ee = a &gt;&gt; 1$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Bitshift Right</td>
<td>$ee1 = 204050 &gt;&gt; a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Shift Right With Sign</td>
<td>$ff = a &gt;&gt;&gt; 1$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Shift Right With Sign</td>
<td>$ff1 = 204050 &gt;&gt;&gt; a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>XOR</td>
<td>$gg = a ^ 4$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>XOR</td>
<td>$gg1 = 4 ^ a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Logical And</td>
<td>$aaa = a1 &amp; &amp; a2'$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Logical And</td>
<td>$aaa1 = a2' &amp; &amp; a1$</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Logical And</td>
<td>$aaa2 = true &amp; &amp; a2'$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Logical And</td>
<td>$aaa3 = a2' &amp; &amp; false$</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Logical Or</td>
<td>$bbb = a2'</td>
<td></td>
<td>a1$</td>
<td>True</td>
</tr>
<tr>
<td>Logical Or</td>
<td>$bbb1 = a1</td>
<td></td>
<td>a2'$</td>
<td>True</td>
</tr>
<tr>
<td>Logical Or</td>
<td>$bbb2 = a2'</td>
<td></td>
<td>false$</td>
<td>True</td>
</tr>
<tr>
<td>Logical Or</td>
<td>$bbb3 = true</td>
<td></td>
<td>a2'$</td>
<td>True</td>
</tr>
<tr>
<td>Logical Not</td>
<td>$ccc = !a2'$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Less Than</td>
<td>$aaaa = a &lt; 2$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Less Than</td>
<td>$aaaa1 = 2 &lt; a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Greater Than</td>
<td>$bbbb = a &gt; 4$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Greater Than</td>
<td>$bbbb1 = 4 &gt; a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Less Than Or Equal</td>
<td>$cccc = a &lt;= 2$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Less Than Or Equal</td>
<td>$cccc1 = 2 &lt;= a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Greater Than Or Equal</td>
<td>$dddd = a &gt;= 4$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Greater Than Or Equal</td>
<td>$dddd1 = 4 &gt;= a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Equals Equals</td>
<td>$eeee = a == 2$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Equals Equals</td>
<td>$eeee1 = 2 == a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Not Equals</td>
<td>$ffff = a'! = 2$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Not Equals</td>
<td>$ffff1 = 2 ! = a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>String Concat.</td>
<td>$aaaaa = a3 + 2$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>String Concat.</td>
<td>$aaaaa1 = 2 + a3$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>String Less Than</td>
<td>$bbbbbb = a3 &lt; &quot;foo&quot;$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>String Less Than</td>
<td>$bbbbbb1 = &quot;foo&quot; &lt; a3$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>String Greater Than</td>
<td>$cccccc = a3 &gt; &quot;foo&quot;$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>String Greater Than</td>
<td>$cccccc1 = &quot;foo&quot; &gt; a3$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>String Less Than Or Equal</td>
<td>$ddddd = a3 &lt;= &quot;foo&quot;$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>String Less Than Or Equal</td>
<td>$ddddd1 = &quot;foo&quot; &lt;= a3$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>String Greater Than Or Equal</td>
<td>$eeeee = a3 &gt;= &quot;foo&quot;$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>String Greater Than Or Equal</td>
<td>$eeeee1 = &quot;foo&quot; &gt;= a3$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Test Case</td>
<td>Description</td>
<td>Passed</td>
<td>Expected</td>
<td>Actual</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------</td>
<td>--------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>Plus Equals</td>
<td>$aaaaaa + = a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Minus Equals</td>
<td>$bbbbbb \ - = a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Times Equals</td>
<td>$cccccc \ast = a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Divide Equals</td>
<td>$dddddd \ / = a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Modulo Equals</td>
<td>$eeeeee % = a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Shift Left Equals</td>
<td>$fffff \ll = a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Shift Right Equals</td>
<td>$ggggg \gg = a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Shift Right With Sign Equals</td>
<td>$hhhhhh \ggg = a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>And Equals</td>
<td>$iiiii &amp; = a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Or Equals</td>
<td>$jjjjj</td>
<td>= a$</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>XOR Equals</td>
<td>$kkkkk \hat{=} a$</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Table 3: Simple Operations Test Results. For cases with multiple operands, an apostrophe(’) indicates which value is tainted.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Description</th>
<th>Passed</th>
<th>Expected</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Object</td>
<td>Untainted construction of simple object</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Simple Object</td>
<td>Tainted object property</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Simple Object</td>
<td>Untainted property of object w/ tainted property</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Simple Object</td>
<td>Tainted return value of object property which uses tainted property</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Simple Object</td>
<td>Untainted object with tainted property</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Simple Object</td>
<td>Untainted name property of untainted Object Pet</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Simple Object</td>
<td>Tainted object</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Simple Object</td>
<td>Property of tainted object</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Simple Object</td>
<td>Untainted prototype of tainted object</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Prototype</td>
<td>Tainted prototype property</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Prototype</td>
<td>Object instance untainted</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Prototype</td>
<td>Shared tainted prototype property</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Prototype</td>
<td>Object instance w/ tainted prototype untainted</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Prototype</td>
<td>Tainted prototype property of object</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Prototype</td>
<td>Return value of prototype method which uses tainted data</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Array</td>
<td>Array after initialization w/ tainted value</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Array</td>
<td>Array access of tainted value</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>In</td>
<td>Result of in operation looking for tainted value</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Array</td>
<td>Joined array contents containing tainted data</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Array</td>
<td>Result of array concatenation</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Array</td>
<td>Element in result of array concatenation</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Array</td>
<td>Result of array slice</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Array</td>
<td>Element in result of array slice</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Array</td>
<td>Associative array access w/ tainted key</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Array</td>
<td>Associative array access of tainted value</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Array</td>
<td>Associative array with tainted key and tainted value</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Table 4: Object Operations Test Results
### Table 5: Scope Operations Test Results

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Description</th>
<th>Passed</th>
<th>Expected</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>With test</td>
<td>'a' untainted in global scope</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>With test</td>
<td>obj.a tainted in global scope</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>With test</td>
<td>'a' tainted in newObj scope</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>With test</td>
<td>One of the vars assigned from the tainted obj property</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>With test</td>
<td>One of the vars assigned from the untainted obj property</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>With test</td>
<td>a remains untainted in the global scope</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Closure Test</td>
<td>First time return value is untainted</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Closure Test</td>
<td>Next time return value IS tainted</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Closure Test</td>
<td>Third time return value is again untainted</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Closure Test</td>
<td>Next time return value IS tainted (and the cycle repeats)</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Private Scoped Props</td>
<td>Setting prop to untainted value</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Private Scoped Props</td>
<td>Setting prop name to tainted value</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Private Scoped Props</td>
<td>Setting prop name back to untainted value</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Function tainted return</td>
<td>Function returns a tainted value</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Conditionals</td>
<td>Dynamically</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

### Table 6: DOM Operations Test Results

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Description</th>
<th>Passed</th>
<th>Expected</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM Write</td>
<td>node's textContent property tainted</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>DOM Write</td>
<td>node's innerHTML property tainted</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>DOM Write</td>
<td>node tainted</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>DOM Read</td>
<td>assigning to node's textContent propagates taint</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>DOM Read</td>
<td>other node's textContent property untainted</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>