Fall 2017

Implementation of Faceted Values in Node.JS.

Andrew Kalenda
San Jose State University

Implementation of Faceted Values in Node.JS.

A project presented to the faculty of the

Department of Computer Science,

in partial fulfillment of

the requirements for the degree

Master of Science,

by

Andrew Kalenda

May 2017.

Abstract

Implementation of Faceted Values in Node.JS

by Andrew Kalenda

Information flow analysis is the study of mechanisms by which developers may protect sensitive data within an ecosystem containing untrusted third-party code. Secure multi-execution is one such mechanism that reliably prevents undesirable information flows, but a programmer's use of secure multi-execution is itself challenging and prone to error. Faceted values have been shown to provide an alternative to secure multi-execution which is, in theory, functionally equivalent. The purpose of this work is to show that the theory holds in practice by implementing usable faceted values in JavaScript via source code transformation. The primary contribution of this project is to provide a library that makes these transformations possible in any standard JavaScript runtime without requiring native support. We build a pipeline that takes JavaScript code with syntactic support for faceted values and, through source code transformation, produces platform-independent JavaScript code containing functional faceted values. Our findings include a method by which we may optimize the use of faceted values through static analysis of the program's information flow.

Acknowledgement

I give my thanks to you, the reader. Whoever you are.

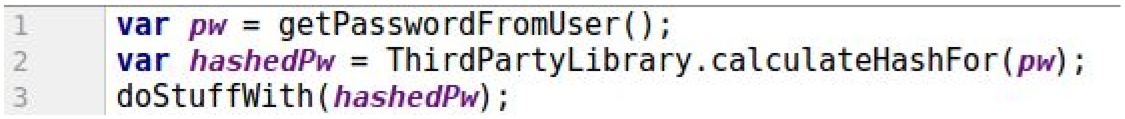
By the simple act of reading this paper, you have justified its existence.

I hope it helps you in some way.

Table of Contents

1: Introduction

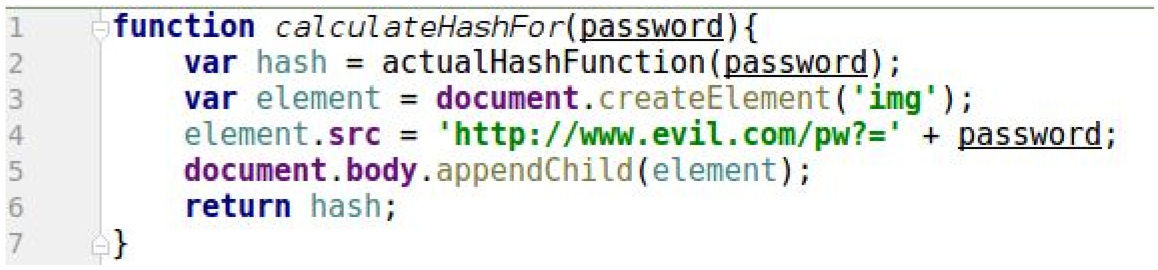
As the the Internet becomes more ubiquitous and integrated into day-to-day life, the growth rate of data online has exploded. Much of this data is sensitive. Passwords, credit card numbers, phone numbers, addresses, browser history, political affiliations, and more can be exploited for reasons commercial, criminal, or both. To exacerbate the problem, the number of potential avenues by which sensitive information can leak also grows. The specific concern which drives the work done in this paper, and its accompanying software, is that web and mobile developers often rely upon libraries from open-source communities such as Github, Bitbucket, NPM, and so on. These third-party libraries can save time and may, in the circumstances, provide security:



```
var pw = getPasswordFromUser();
var hashedPw = ThirdPartyLibrary.calculateHashFor(pw);
doStuffWith(hashedPw);
```

Listing 1.1: A JavaScript program using a fictional third-party library.
It operates on the assumption that the third-party library provides a secure hashing algorithm.

However, a third-party library may also contain malicious software hidden within itself, within the library's own dependency tree, or cunningly injected through other means:



```javascript
function calculateHashFor(password){
    var hash = actualHashFunction(password);
    var element = document.createElement('img');
    element.src = 'http://www.evil.com/pw?=' + password;
    document.body.appendChild(element);
    return hash;
}
```

Listing 1.2: A simple example of malicious software.

The internals of the third-party library send sensitive information to a website for collection. This exploitation is rudimentary enough that a security-minded developer can quickly spot it when reviewing the library, but that is not always the case. Therefore, we desire a means by which to algorithmically deny sensitive information to these kinds of programs.

Our objective is to demonstrate *faceted values* as a viable information flow control in order to prevent sensitive information from being leaked to unwanted observers. *Information flow* is the means by which data and metadata may propagate through a computer program; these flows, and how sensitive information may be leaked, are explained in Section 2.1.

Our primary contribution is a JavaScript library to which a developer may give their own JavaScript program, which has been marked with *faceting* information. The faceting information results in sensitive data and attached metadata flowing through the program in tandem. In this way any data that is a consequent of the sensitive information necessarily becomes sensitive to some extent, and decisions on how to present data to varied observers can be made:

```
cloak k {
    var pw = getPasswordFromUser();
    var hashedPw = ThirdPartyLibrary.calculateHashFor(pw);
    doStuffWith(hashedPw);
}
```

Listing 1.3: User-created code marked with faceting information.

The code from Listing 1.1 has been modified to attach metadata labelling sensitive information. The cloak keyword, followed by a label, signifies that data created within the subsequent code block is classified according to the label k.

```
var FacetedValuesJS = require('faceted-values-js');
FacetedValuesJS
    .fromFile('listing1.1.3.js')
    .toFile('listing1.1.5.js');
```

Listing 1.4: A straightforward use of our library.

The code shown in Listing 1.3 is not valid JavaScript ECMAScript6 syntax and will not function as-is. Using the library, we take that code as input and write it out as functioning JavaScript, as seen in Listing 1.5.

```
var FacetedValuesJS = require('faceted-values-js');
var FacetedValue = FacetedValuesJS.FacetedValue;
var Cloak = FacetedValuesJS.Cloak;
var view = [];
view.push('k');
{
    var pw = Cloak(view, getPasswordFromUser());
    var hashedPw = Cloak.invokeFunction(ThirdPartyLibrary.calculateHashFor, this, [pw]);
    Cloak.invokeFunction(doStuffWith, this, [hashedPw]);
}
view.pop();
```

Listing 1.5: The code from Listing 1.3, rewritten.

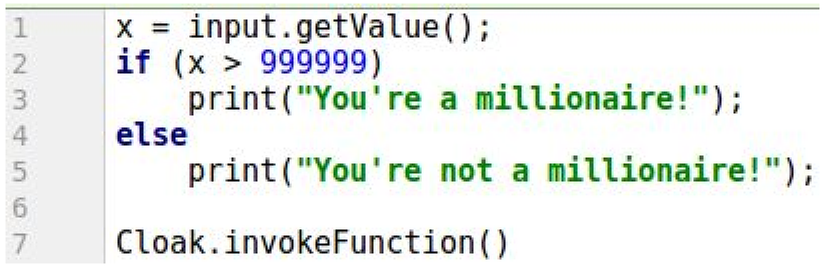
It has been parsed into an abstract syntax tree refactored to use faceted values, and written into code. The sensitive password information is hidden within a proxy object and labeled with the k classification. The hidden information is presented only when the classification of the observer viewing it is the same as the classification of the data itself. In this way, the data is kept safe even when passed to the malicious software. E.g. only an observer labeled as k will see the password labeled as k. A more thorough explanation is given in Section 2.4.

Even so, this is not a freestanding solution. Referring back to Listing 1.2, there is still the matter of how to handle the transmission of data to “evil.com.” If our library is acting on its own, then `Cloak.invokeFunction` unpacks the facets of `pw`, invokes the hash function on each, and reassembles the returned values into a new faceted value. The result is that both the private and public value are sent to `evil.com`. To prevent this, faceted values must be accompanied by, and in communication with, a technology that acts as gatekeeper – treating system output calls such as HTML requests as observers, and setting views accordingly. Unlike our faceted values library, such a technology would vary greatly depending on its environment. There would likely be individual implementations specific to Node.js, Google Chrome, Mozilla Firefox, and so on, all of which remain as future work. These “companion libraries” are necessary for faceted values to provide the needed protection, and having them are the greatest obstacle to our approach being used in practice.

2: Background

2.1: Information flows and leaks

Consider the following code:



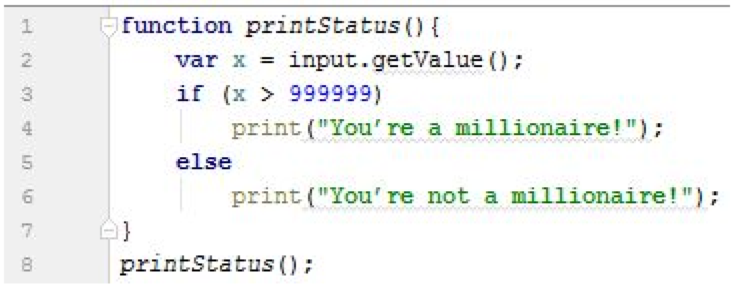
```
x = input.getValue();
if (x > 999999)
    print("You're a millionaire!");
else
    print("You're not a millionaire!");

Cloak.invokeFunction()
```

Listing 2.1.1: A simple demonstration of information flow.

There is an *explicit flow* of information from `input` to ***x***. There is also an *implicit*[1] *flow* of information from ***x*** to the `print` statement. These flows reflect the intended function of the program, and are not themselves problematic.

As simple as this code is, it has a security vulnerability in a certain context. The programmer, in not having used Javascript's **var** keyword, has made ***x*** global. If malicious code can be inserted into the same scope as this code — a plausible enough scenario in web browsers — the value of ***x*** can be read. Thus there is a "dangling" information flow from this program to hypothetical third-parties. A conscientious programmer may then change the code to the following:



```
function printStatus(){
    var x = input.getValue();
    if (x > 999999)
        print("You're a millionaire!");
    else
        print("You're not a millionaire!");
}
printStatus();
```

Listing 2.1.2: Closing a dangling information flow.
The flow from x is now closed because it is safely ensconced within the scope of the `printStatus()` function.

However, a more subtle leak yet remains. The output of the print statement is determined by *x*, and therefore partial information about *x* can be drawn from which string is printed. To clarify, if **`"You're a millionaire!"`** is printed, viewers know that the value of *x* was at least one million. This is also an implicit flow.

We can close this implicit flow by making different data available to different viewers depending on their authorization level. For example, the intended viewer of *x* in the code above

[1] Note that here the word *implicit* is meant in a mathematical sense. The flow of information is due to logical consequences. It does not involve guesswork, and does not carry *implicit*'s alternate meanings of *hinted*, *insinuated*, or *intimated*.

may see either `"You’re a millionaire!"` or `"You’re not a millionaire!"`, but an unintended user may simply receive a garbage string such as `"Invalid user"` or `"403 Access Forbidden"`. We may even show the unauthorized user nothing at all, or go so far as to crash the program in an effort to protect its secrets.

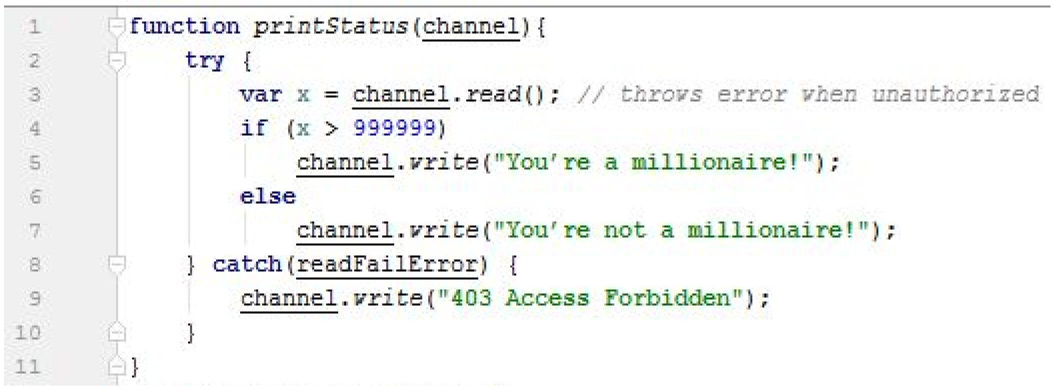
The objective of faceted values is to provide these different views upon data in such a way that no implicit or explicit flows can be found, and it is the objective of this paper to demonstrate that faceted values can work in practice.

2.2: Secure multi-execution as a solution

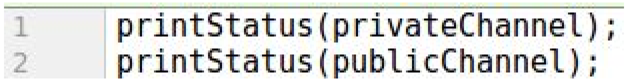
We now provide the context in which faceted values were invented.

Secure multi-execution is an information flow security measure. A given application has security levels or aspects associated with it. For each of these, the program executes once. Each of these executions is part of a *channel*, from which different observers — authorized for different levels, or privy to different aspects — may receive different data. [10]

Let us explain this in more detail, using a particular styling of security levels. Consider again Listing 2.1.2. There is an information leak: The printed string reveals information about the private variable x. To seal this leak, we want to have different views presented to different observers. We can accomplish this by executing subroutines multiple times for different authorization levels. When a given datum does not meet the current authorization level, a default or garbage value is used instead. The output of the execution is sent through a communication channel according to the authorization level. Finally, a user’s authorization level determines which channel they receive data from.



```
function printStatus(channel){
    try {
        var x = channel.read(); // throws error when unauthorized
        if (x > 999999)
            channel.write("You're a millionaire!");
        else
            channel.write("You're not a millionaire!");
    } catch(readFailError) {
        channel.write("403 Access Forbidden");
    }
}
```



```
printStatus(privateChannel);
printStatus(publicChannel);
```

Listing 2.2.1: Output abstracted into two communication channels.
The program, encapsulated within the printStatus function, executes twice: once for each communication channel. Any observer will see *only one* of these channels according to their authorization level.

Each channel is associated with an authorization level, which is in turn associated with a set of observers. When the data is viewed through the lens of privateChannel, they will see the authentic information regarding whether or not they are a millionaire. When the data is viewed through publicChannel, they will receive the message `"403 Access Forbidden`" and nothing more. This message could also be a counterfeit message, such `"You're not a millionaire!"`, which has the advantage that its being indistinguishable from an authentic message means that the observer is unaware of the fact that they have been given counterfeit data.

Strictly speaking, secure multi-execution can have any number of channels. Yet at any given point, it is most practical to think of the channels in binary terms: An observer of data is either privy to its true nature or not depending on some aspect of authorization. These aspects could be labelled as "loggedInUser," "superUser," "admin," "instructor," "grader," "banker," "superSpy," "grandma," "POTUS," "pancakeKing," or any other string at the developer's discretion.

We can then organize our communication channels into a binary tree based on these aspects:

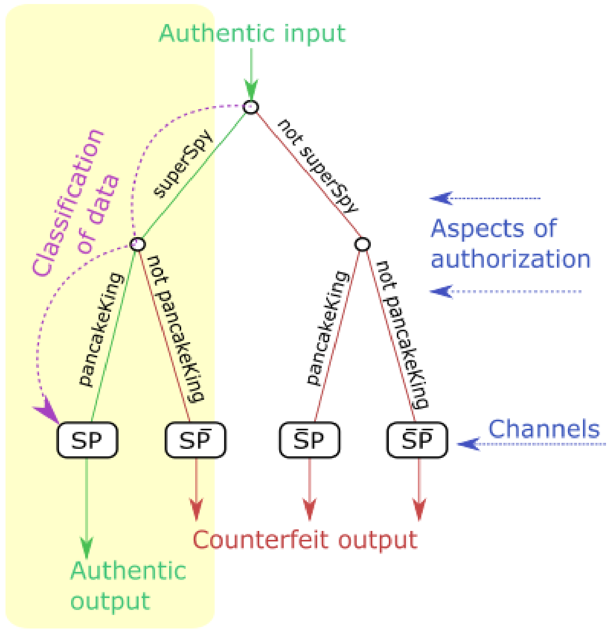
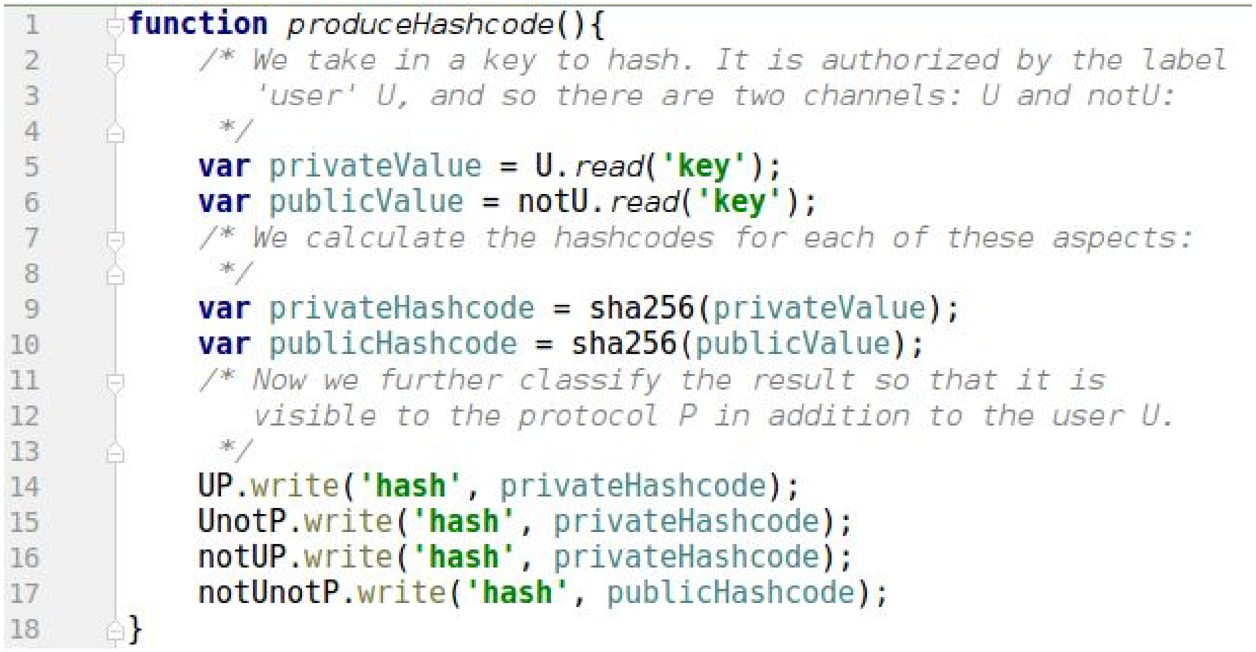


Figure 2.2.2: Secure multi-execution data flow.
Classification of data bifurcates according to aspects of information.

Each aspect bifurcates possible flows of information into two channels. Then, the process of changing a datum's classification is very simple: Simply determine what aspect reflects the nature of the change, and direct its flow down that channel to match.

For example, suppose that we wish to take a user's password and produce a hashcode that can be matched against that user on future logins. The password itself is not going to be available to the system; therefore we may label at with an aspect "userPrivate", U. The hashcode for the password would, by default, be produced with the same aspect. However, we want to make the hashcode available to the password management system, and so arbitrarily give it a label "password", P. Then, it can be sent along channels UP, $U\overline{P}$, and $\overline{U}P$, but not $\overline{U}\,\overline{P}$. This reclassifies the data so that the hashcode is not just available to the user, but also the protocol. Yet, it still must be one or the either:



```
function produceHashcode(){
    /* We take in a key to hash. It is authorized by the label
       'user' U, and so there are two channels: U and notU:
     */
    var privateValue = U.read('key');
    var publicValue = notU.read('key');
    /* We calculate the hashcodes for each of these aspects:
     */
    var privateHashcode = sha256(privateValue);
    var publicHashcode = sha256(publicValue);
    /* Now we further classify the result so that it is
       visible to the protocol P in addition to the user U.
     */
    UP.write('hash', privateHashcode);
    UnotP.write('hash', privateHashcode);
    notUP.write('hash', privateHashcode);
    notUnotP.write('hash', publicHashcode);
}
```

Listing 2.2.3: A styling of secure multi-execution.

The data in this function are classified by a set of two aspects: user privacy (U) and the needs of a protocol (P). The power set of {U, P} lists the channels we need. In this particular function, only one of those channels lacks authorization to read the hashcode.

2.3: The problem of usability

Information flows have a serious problem in terms of usability.

Consider the binary tree of communication channels. A developer, and their program, needs to contend with a large number of communication channels — and the number grows at an exponential rate depending on how many relevant facets they come up with! They may be able to “prune” the tree by finding and removing redundant branches of channels, but this risks human error.

Pragmatically speaking, it is more likely that due to time constraints, absent-mindedness, ignorance, human error, or some other (perfectly understandable) reason the developer is not going to create a complete tree of channels and then prune it correctly. It is more likely that they will simply create channels ad hoc.

A developer must also remain cognizant of the side effects of multiple executions, particularly when the outcome of these executions depend upon the order in which they run. This is most problematic when the multiple executions occur on a function-by-function basis. A reliable way to avoid the issue is to execute the program in its entirety once for each channel without shared state, but even when this is possible the performance cost is great.

This leads to further problems with classification and declassification of data, where the developers may take shortcuts and/or reuse existing channels. Developers are at risk of not using them correctly and sabotaging their own efforts. This is not necessarily a failing on the part of the developers. Developers who contend with the sanctity of data are already hard-pressed: They must be continuously mindful of vulnerabilities and best practice, because the cybersecurity landscape is dynamic, ever-changing, and assiduous.

We want an alternative that is self-managing and easy-to-use, and are willing to sacrifice a certain amount of flexibility to purchase as much.

2.4: What are faceted values?

A *faceted value* is so-called because it is in essence a single value with many facets, and the facet that is visible — e.g. the value that it presents — is dependent on the context in which the observer resides.

A faceted value contains a binary tree of immutable values wherein each branch is determined by membership in an arbitrary[2] label[3]. The basic format follows:

[2] Labels are selected by whomever writes the code for the faceted values. As an example, a developer programming a recipe management system may arbitrarily choose the label "pancakeKing" to classify secret ingredients.

[3] In theory, the label can be any value of any type which supports an equality comparison. In practice, strings are the type of choice for labels.

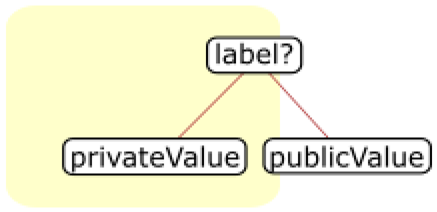


Figure 2.4.1: The basic form of a faceted value.
In text: `<label ? privateValue : publicValue>`

Its resemblance to classic ternary operations is not coincidental. While a faceted value is a data structure instead of an expression, it evaluates to a single nonfaceted value in a similar way. Of the facets (publicValue and privateValue), one may appear to an observer based on a simple question: “Does the observer’s view contain this label? If so, present privateValue. Else, present publicValue.” To clarify even further, this directly parallels the classification scheme presented in Figure 2.2.2. The observer’s view indicates their own classification, and the labels within the faceted value indicate the classification of its data. On being viewed the faceted value resolves to one of its facets based on how the classifications of data and observer match or differ. If the faceted value is observed in a context that matches its classification — that is, the “current view” has a matching label — then the classified facet (or private value) is presented. Otherwise, the unclassified unfacet (or public value) is presented.

The private and public values can be of any conceivable type. In particular, the values can themselves be faceted values:

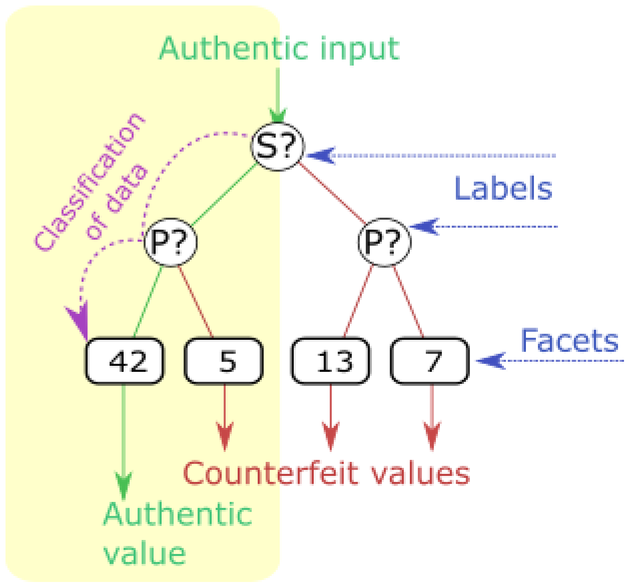
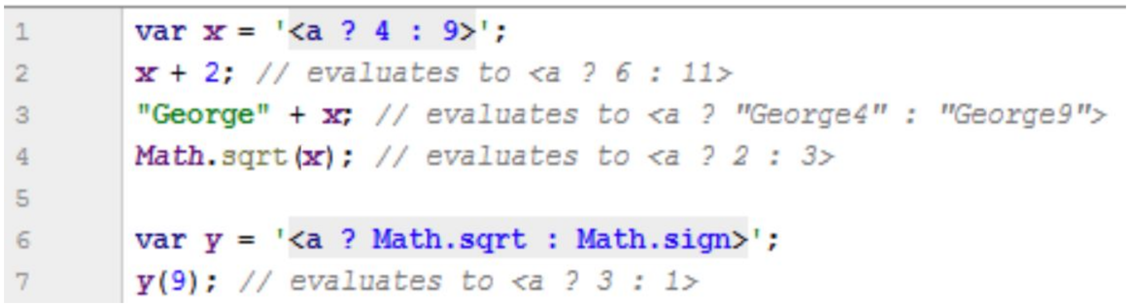


Figure 2.4.3: A nested, faceted value with four facets.
In text: `<S ? <P ? 42 : 5> : <P ? 13 : 7>>`

Note the similarity between the secure multi-execution pipelines diagramed in Figure 2.2.2 and the structure of the faceted value in Figure 2.4.3. It is by design, not coincidence. Faceted values are meant to be functionally equivalent to that particular styling of secure multi-execution and, by extension, are able to provide the same protections.

One of the key points with faceted values is that their composability is very well-defined. If one were to, say, add two to a faceted value, it would add two to all of its values. Consequently, any operation that can be performed on the values within the faceted value can be performed on the faceted value itself, and vice-versa:



```
var x = '<a ? 4 : 9>';
x + 2; // evaluates to <a ? 6 : 11>
"George" + x; // evaluates to <a ? "George4" : "George9">
Math.sqrt(x); // evaluates to <a ? 2 : 3>

var y = '<a ? Math.sqrt : Math.sign>';
y(9); // evaluates to <a ? 3 : 1>
```

Listing 2.4.4: Examples of different operations involving faceted values in various roles.

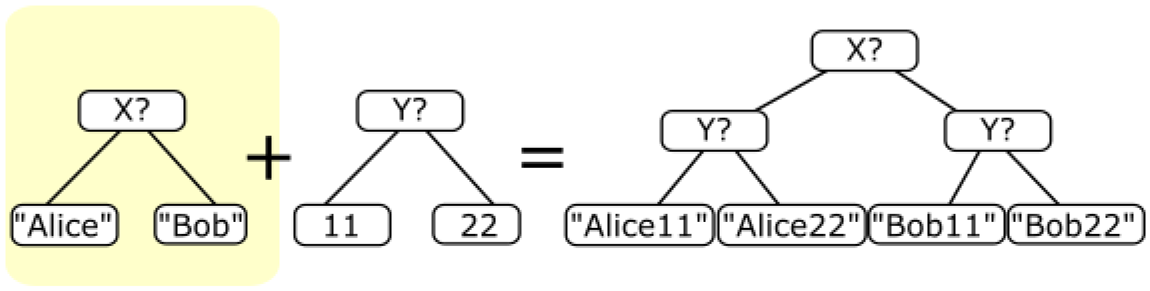


Figure 2.4.5: Products of faceted values.

When operations include multiple faceted values as operands, the classification of data remains consistent. Sometimes the result is a more complex, nested faceted value.

2.5: Related work

In prior work [6] it has been shown through prototyping that faceted values cleanly handle implicit information flows; can neatly be declassified; are formalized in the idealized, untyped imperative "λ^{facet}" language and can migrate thence to concrete programming languages; can be prototyped in Firefox's JavaScript runtime; can protect against a Cross-Site Scripting attack; and are especially promising in terms of performance compared to secure multi-execution [6]. It has also been shown that faceted values can be implemented as monads in the statically, strongly typed language Haskell with no need for native support [9]. Lastly, their relation to specific information security issues have been further elucidated in terms of data declassification, exceptional behavior, and label-based access controls [13].

Let us revisit the topic of Secure Multi-Execution (SME). SME may provide information flow security as described in Section 2.2, but it is worth noting that it is not limited to the bifurcation approach depicted in Figure 2.2.2. The channels chosen could reflect upon any conceivable system of classification, binary or not. The question one ask must be, “What is most appropriate to my needs?” Diverse peoples have contributed to SME technology. Rafnsson and Sabelfeld [10] provide a great deal of information on the level of a survey with exceptional readability. In it, they touch upon this question by devoting a section to the “Pros and cons of secure multi-execution”, by which we may compare SME with faceted values. Firstly, SME has strong guarantees for non-interference: different classifications of data are segregated into separate executions; in faceted values, data of different classifications march through the program’s execution hand-in-hand. Secondly, SME depends on the runtime environment’s process scheduling, which faceted values have no innate concern for. Thirdly, SME has issues with declassification of data, which faceted values do not.[4] Fourthly, SME has transparency issues with non-deterministic evaluations, whereas in faceted values a programmer could opt to use either the same or different non-deterministic values across all facets at their own discretion.[5] On the other hand, both SME and faceted values share a few shortcomings. They risk situational logical errors on the developers’ part due to code segments being executed multiple times and “stacking” their side effects.[6] They repair the damage caused by information flow leaks instead of protecting against breaches. Similarly, both prevent the damage silently

[4] Declassification of facets is accomplished simply by collapsing the branches decided by a given label into a single facet. For example, consider Figure 2.4.5. A developer may decide that in the product on the right, the Y label no longer matters. They may elect to declassify Y, so that the private values are now public, resulting in a faceted value `<X ? “Alice11” : “Bob11”>`.

[5] For example, if readouts from a Geiger counter are being used to immediately add non-deterministic numbers to a faceted value with eight facets, a single readout could be added to all eight facets. Or, eight readouts can be added to single facets. In SME, only the latter is an option.

[6] This can occur when facets need to be evaluated for a function that cannot handle faceted values internally. The facets are unpacked and executed separately, causing multiple executions similar to SME’s on a smaller scale.

and do not have means by which to identify attacks, although both could potentially be modified to do so. Lastly, both by their very nature demand a marked increase in computational resources.[7]

Another set of relevant and interesting technologies can be found in Object Capability Models (OCM).[8] These are not branded as information flow controls, but much like faceted values they are built in response to untrusted third-party code in a web programming setting. Unlike faceted values, they seek not to repair the damage done by malicious third parties but to prevent it by introducing an intermediary between trusted and untrusted realms of data and program execution. Where before a trusted and untrusted program construct would comprise an object-subject interaction, now the OCM acts as both subject of one and object of the other. In this way each program and datum — whether the first-party of the developer, the second-party of the web browser, or the third-party of external libraries — can be given an authority level commensurate with its provenance, and thereby may be authorized or not to access data or behaviors. OCM systems have vulnerabilities of their own[2], but an impressive array of works have built off the technology since then.

It is worth noting that while these technologies may compete they are not mutually exclusive. It is entirely plausible for a developer to join them together. Insecurities in web programming are like an amorphous beast: insidious, adaptive, and never fully understood. To have many layers of defenses, each with its own strengths, is advisable if not always feasible in terms of time and resources. However, such a joining of these technologies is not the purpose of this project. Our purpose is to assist in the development of faceted values as a viable candidate.

---

[7] Depending on the situation, either SME or faceted values may have a greater performance impact in processing time or memory. In most situations, faceted values appear to have the advantage, particularly when libraries are drawn from many domains with little intermingling among them. [6]

[8] A widely recognizable example of OCMs would be the Google Caja Compiler.

3: Project

3.1: Goals

Our ultimate objective is to see faceted values become a natively supported feature of JavaScript runtimes. First, we realize the functionality of faceted values by creating a library, FacetedValue.js, that does not require native support in a JavaScript interpreter or compiler. We then give it syntactic sugar to demonstrate its ease of use. This syntactic sugar emulates the look and feel of native support, and provides a proof of concept for native JavaScript implementations to include their own faceted values.

3.2: Why JavaScript?

JavaScript lends itself well to the kind of metaprogramming needed in this project, but there is a much simpler reason it was chosen: The vulnerabilities in information flow that Faceted Values seek to address are most common when using JavaScript.

To elaborate, the vulnerabilities are most commonly found when web browsing. Whether by way of a browser plugin, a trojan cleverly inserted into a webpage's advertisements, or a virus in its JavaScript libraries, a website's classified data may be vulnerable to the kinds of observations described in Section 2.1 of this paper.

It has already been shown that faceted values can be implemented in JavaScript — a proof of concept implementation based on Narcissus [6]. Narcissus is a JavaScript interpreter designed for rapid prototyping of language design ideas, and is not intended for production (or performant) use. Our work expands on this prior implementation by rewriting source code to bring the functionality of faceted values to all varieties of JavaScript.

JavaScript development has its own perks that we take advantage of. JSDoc is a tool that generates API documentation for JavaScript code. Even if not using the tool itself, JSDoc compliancy suggests conventions that can greatly clarify how code works and expose flaws in one's code. This project features extensive use of JSDocs.

Unit testing is accomplished using the Nodeunit library[9]. The general strategy for these unit tests is to traverse the code paths in the library and demonstrate the Listings in this paper.

3.3: AST-Query for syntactic sugar

One of our stated goals was to "give [faceted values] syntactic sugar to demonstrate [their] ease of use." This reflects upon the problem of usability described in Section 1.5 of this paper. The task of this project is not simply to provide a functioning library for faceted values, but to make it usable by providing a solution that will take accessible but non-functioning code provided by a developer (as seen in Listing 3.3.1.1) and produce less accessible but functioning code ready for a compiler (as in Listing 3.3.1.2).

[9] NodeUnit can be found at https://github.com/caolan/nodeunit

```javascript
var x = '<a ? 4 : 9>';
x + 2; // evaluates to <a ? 6 : 11>
```

Listing 3.3.1.1: An excerpt from the earlier Listing 1.6.4.
This demonstrates a natural means of expressing the faceted value, <a ? 4 : 9>.

```javascript
var x = new FacetedValue('a', 4, 9);
x.binaryOps('+', 2);
x.toString(); // evaluates to <a ? 6 : 11>
```

Listing 3.3.1.2: The prior Listing implemented using FacetedValue.js.
This is a functional means of expressing faceted values, but it does not provide good usability.

We considered two technologies, SweetJS[10] and AST-Query[11], and ultimately selected AST-Query. The reason we did so was because faceted values can be written as simple string literals. This means that the source file we take as input can be entirely valid JavaScript code which parses into a complete abstract syntax tree, and we can refactor the abstract syntax tree to produce a new program instead of directly rewriting the text of the source code.

Regarding SweetJS, it is a JavaScript library that allows its user to define hygienic macros that, in effect, extend the JavaScript programming language. It maintains sufficient history while lexing a given program so as to separate lexing from parsing, and is thus able to intervene between the two and rewrite the program according to the macros given [8]. Hypothetically, it is a perfect match for what needs to be done with faceted values. In practice, SweetJS is very difficult to use. This is because it is exceptionally powerful and flexible; it combines both lexical and syntactic parsing of code which may not produce a valid JavaScript program and rewrites it into a fully functioning JavaScript program. However, this power and flexibility exceeded our own needs, and we opted for the simpler approach, which will be described in Section 3.4.3.

[10] SweetJS can be found at https://www.sweetjs.org/
[11] AST-Query can be found at https://github.com/SBoudrias/AST-query

3.4: Program overview

The program is being developed in the Node.js JavaScript runtime. The code base is expansive enough that covering it within this paper would be impractical. It is hosted on a Git repository, which can be found at https://github.com/akalenda/FacetedValuesJS. Actual files and directories notwithstanding, the project is conceptually structured in a few broad parts:

3.4.1: FacetedValue.js

This file is the core of the library. It provides all of the methods necessary to use faceted values and guarantees their various properties (such as referential transparency, monadism, and so on).

What it does not provide is a good user experience (as explained in Section 2.3). Its concern is purely to function, and function correctly. To that end, `testFacetedValue.js` exists within the test directory. It performs a (near-exhaustive) battery of unit tests that, if passed, guarantee that the faceted values are working correctly. Moreover, these tests provide many working examples of the library in action.

FacetedValue objects are referentially transparent in the sense that any primitive operation involving them either as operand or operation is applied to all of its facets. This is accomplished by interceding in the operation and deferring it to FacetedValue-class methods. The source code transformation of simple operations into more complex object-managed operations is styled after similar operations on “virtual values” by Kannan et al in 2016 [12], as were formalized in 2011 by Austin et al [3].

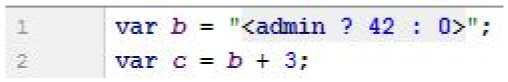
3.4.2: Cloak.js

The weakness of `FacetedValue.js` is that it does not by itself provide the full protection against unprivileged observers we desire. While it correctly handles the flow of classified and unclassified facets within a single value, it does not present different views to different observers (as described in Section 1.3). This was intended for two reasons. Firstly, while such protections are integral to the role of faceted values in information flow security, they are separate from faceted values in and of of themselves. Secondly, we do not want our values to be obfuscated while debugging the source code transformation pipeline. Therefore we have extracted the actual concealment of facets into Cloak, which is a wrapper class that encapsulates the functionality of faceted values within a proxy object. The proxy objects are those proposed by Cutsem et al in 2010 [1] and clarified in 2013 [7], thereafter to be accepted into the ECMAScript 6 specifications[11] and implemented in a variety of JavaScript runtimes.

3.4.3: FacetedValuesJS.js

This is the most experimental portion of the code base, as attested by its having the least organized code in the project. This file draws together `FacetedValue.js`, `Cloak.js`, and the third-party library `ast-query` to refactor programs through a unified front. The library also presents itself through `FacetedValuesJS.js` as an NPM module[12]. In broad strokes, it computes as shown in Figures 3.4.3.2 through 3.4.3.4:

[12] The NPM module can be found at https://www.npmjs.com/package/faceted-values-js . However, it is secondary to the Github repository found at https://github.com/akalenda/FacetedValuesJS .



```
var b = "<admin ? 42 : 0>";
var c = b + 3;
```

Listing 3.4.3.1: A simple program containing faceted values.

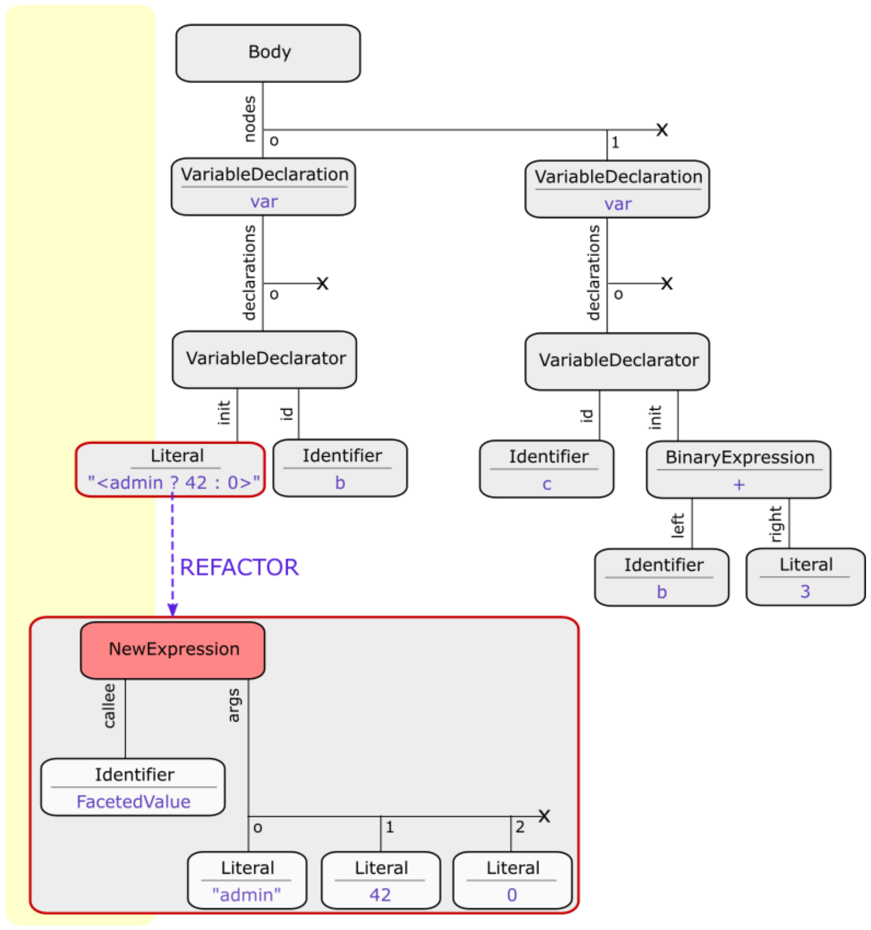


Figure 3.4.3.2: Faceted value substitution in an abstract syntax tree.

A tree is produced from the previous Listing 3.4.3.1. It is then scanned for nodes which are string literals that can be interpreted as faceted values (outlined in red). That literal is then replaced with a call to the `FacetedValue` constructor, and the node is marked as faceted (shown in red).

On that abstract syntax tree we overlay a graph of information flows from one part of the program to another. From the values that are declared to be faceted, we follow the information flows throughout the program to determine where faceted values may arrive, as shown in Figure 3.4.3.3:

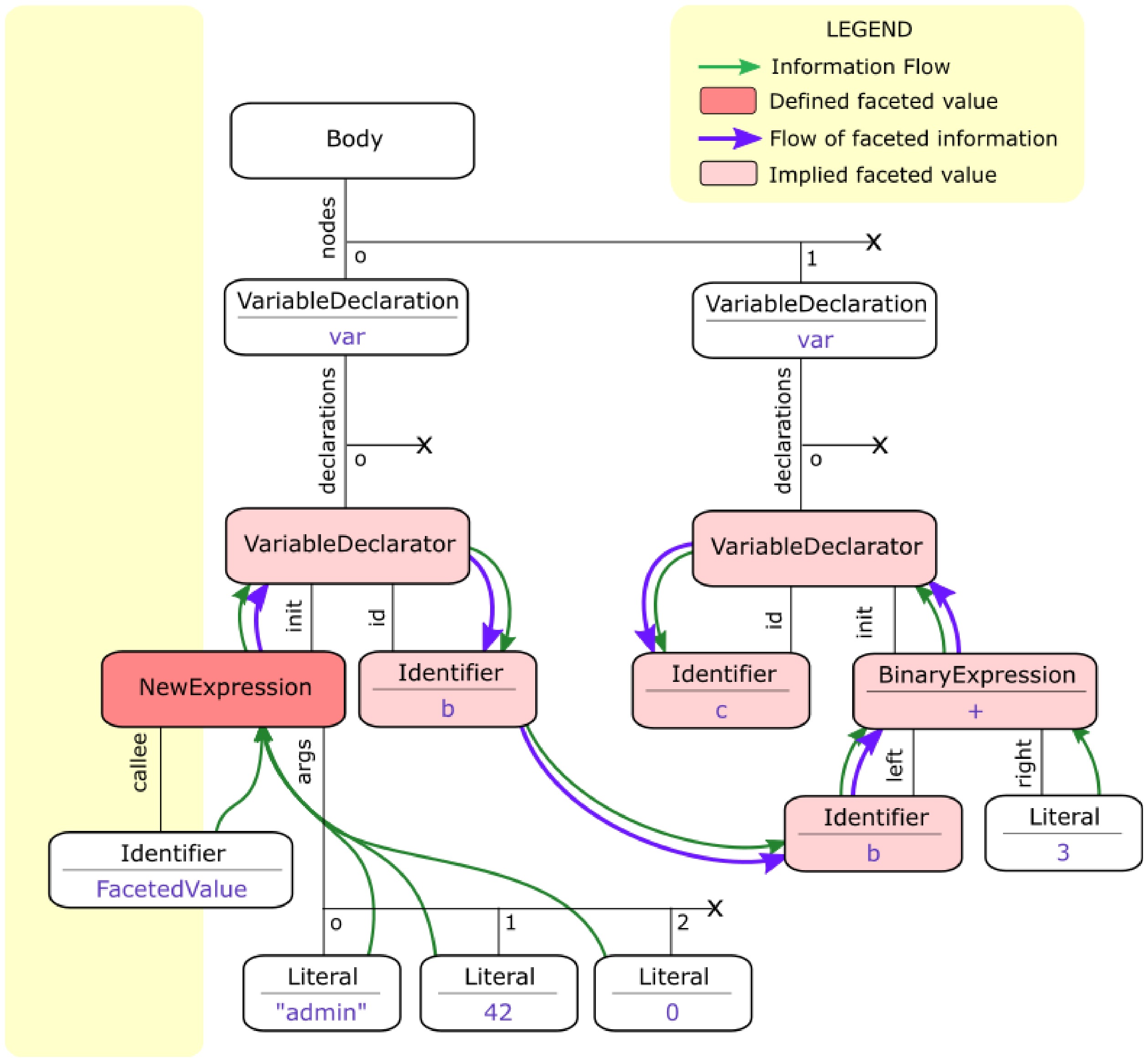


Figure 3.4.3.3: Information flows through the abstract syntax tree.
This continues from the previous Figure 3.4.3.2. The information flows (green arrows) are represented in the program by the `node.outgoingFlows` field. The implied faceted values (pink nodes) are represented in the program by the `node.faceting` field. The purple arrows illustrate which information flows conduct faceted information.

We then refactor the abstract syntax tree to accommodate the faceted values. Each unique node type (i.e. AssignmentExpression, BinaryExpression, etc.) is individually considered for how it may or may not need to be refactored based on that node's operational semantics. Our example has one such operation, and its refactoring is shown in Figure 3.4.3.4:

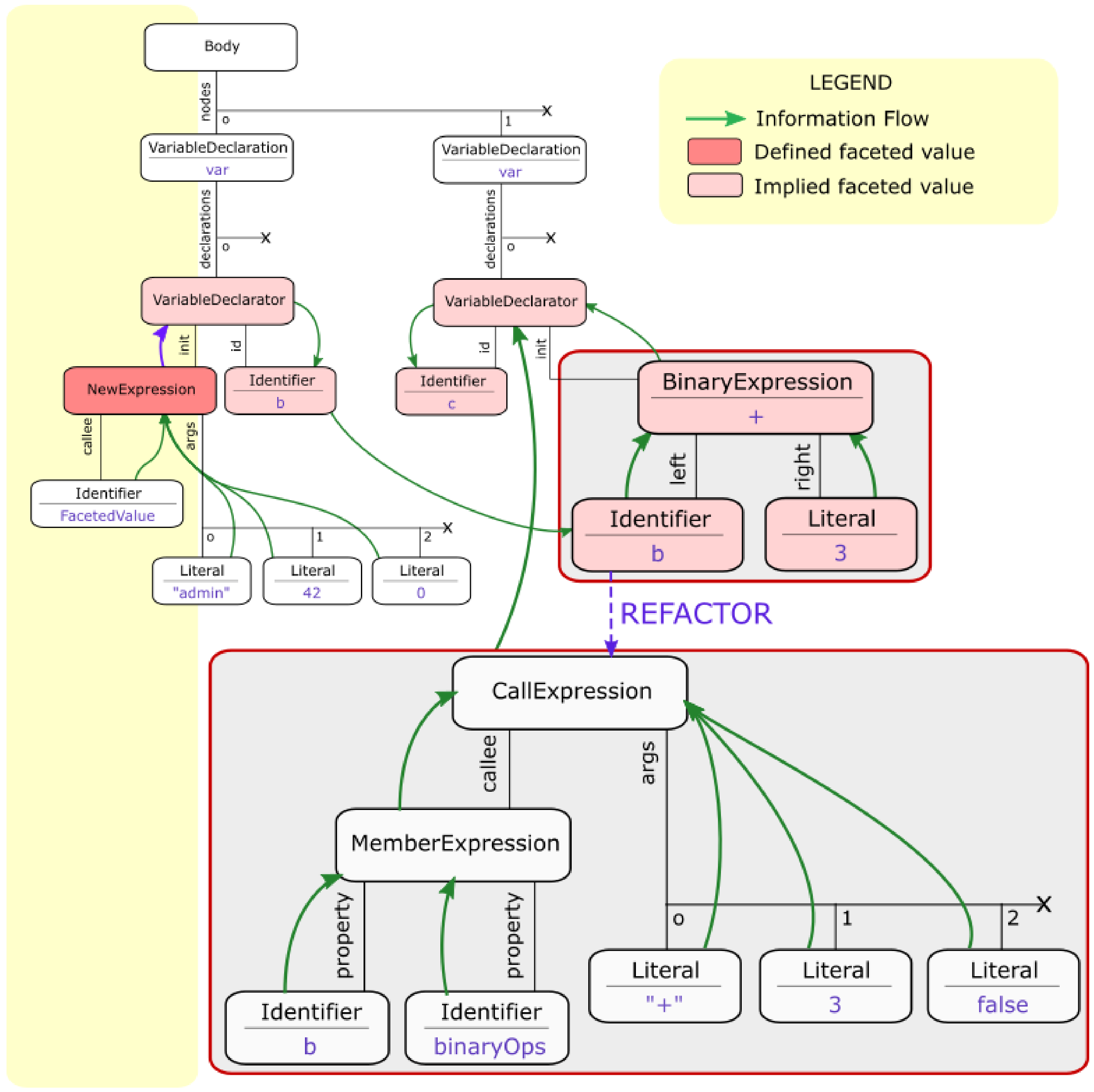


Figure 3.4.3.4: An operation refactored to accommodate faceted values.

This continues from the previous Figure 3.4.3.3. The tree is scanned for nodes that are marked as being a conductor for faceted values. Shown at the bottom is the abstract syntax tree fragment the `BinaryExpression` refactors into.

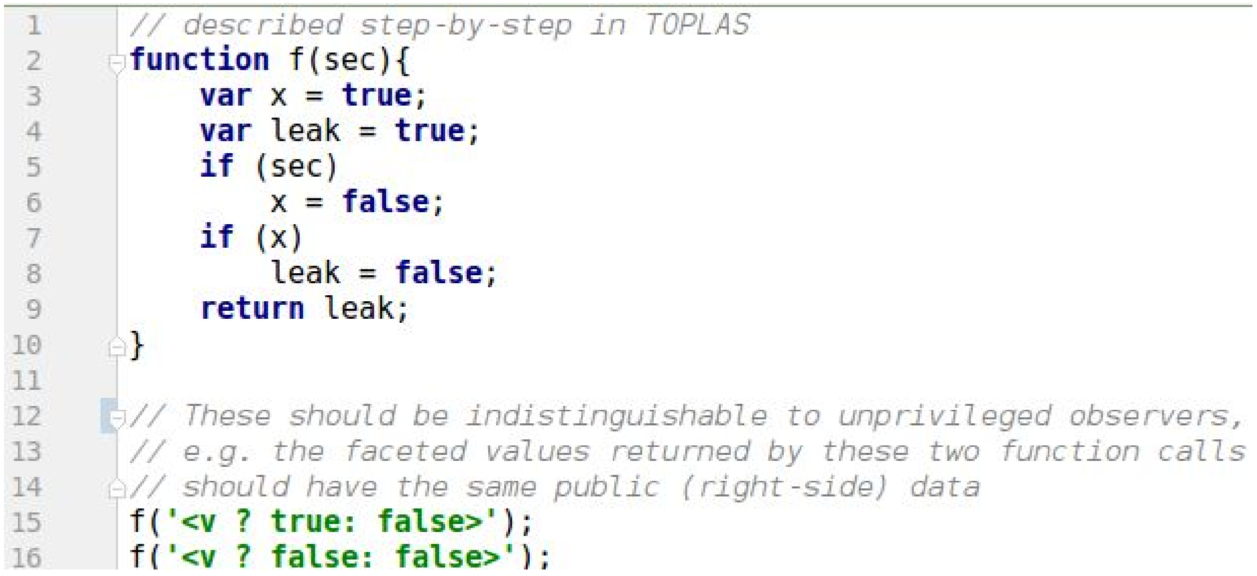
```
var FacetedValue = require("FacetedValue.js");
var b = new FacetedValue("admin", 42, 0);
var c = b.binaryOps("+", 3, false);
```

Listing 3.4.3.5: Final output of the source code-transformation pipeline.

4: Results

4.1: A Faceted Value pipeline

Stated as a design goal in Section 3.1, we set out to provide a proof of concept for faceted value implementations. Although our library by no means handles every possible use case for faceted values[13], it provides a complete pipeline of source code transformation that suffices to demonstrate their intended use and function.



```javascript
// described step-by-step in TOPLAS
function f(sec){
    var x = true;
    var leak = true;
    if (sec)
        x = false;
    if (x)
        leak = false;
    return leak;
}

// These should be indistinguishable to unprivileged observers,
// e.g. the faceted values returned by these two function calls
// should have the same public (right-side) data
f('<v ? true: false>');
f('<v ? false: false>');
```

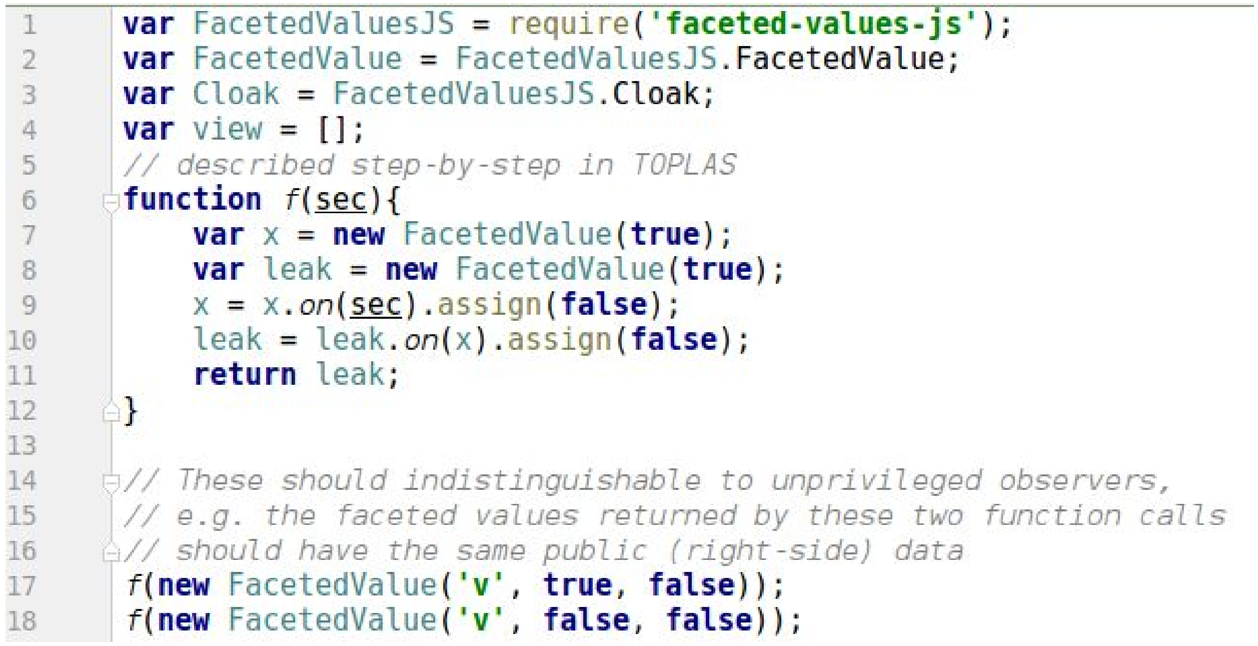
Listing 4.1.1: Faceting-specific input source code.

The function `f` exposes many issues with information flow security. It contains pitfalls that an information flow control must avoid, as ours does. Its vulnerabilities are detailed by Austin et al [13].

Users need to be able to write programs that use faceted values in a natural way, such as is seen in Listing 4.1.1. However, without native support for faceted values in the Javascript

[13] When the library encounters a known use case that it does not yet handle correctly, it throws an error. There may, however, be unknown use cases that are also not handled, and for these the trans-compiler may complete without error.

runtime, this is not functioning code. Therefore we must provide an intermediary program that refactors the code into something more functional, as in Listing 4.1.2.



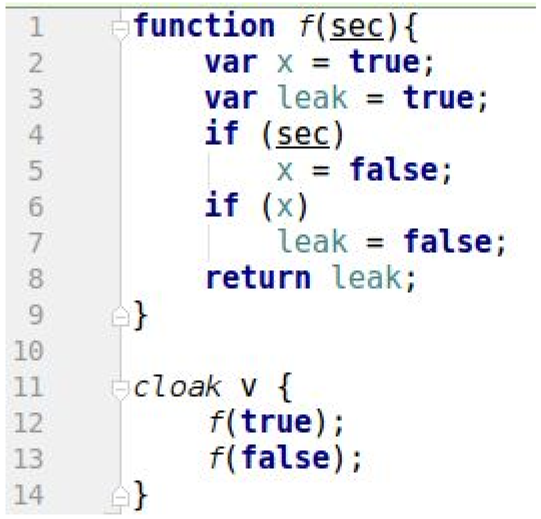
```javascript
var FacetedValuesJS = require('faceted-values-js');
var FacetedValue = FacetedValuesJS.FacetedValue;
var Cloak = FacetedValuesJS.Cloak;
var view = [];
// described step-by-step in TOPLAS
function f(sec){
    var x = new FacetedValue(true);
    var leak = new FacetedValue(true);
    x = x.on(sec).assign(false);
    leak = leak.on(x).assign(false);
    return leak;
}

// These should indistinguishable to unprivileged observers,
// e.g. the faceted values returned by these two function calls
// should have the same public (right-side) data
f(new FacetedValue('v', true, false));
f(new FacetedValue('v', false, false));
```

Listing 4.1.2: Faceting-specific output source code.
It is the previous Listing 4.1.1, converted into functioning JavaScript that controls the flow of classified information.

This code uses the FacetedValue class with methods to accept expressions and execute them in a way that preserves the flow of classified information. It is similar to how the code may be interpreted in a native implementation of faceted values. However the contents of the faceted value are not concealed from a third-party malicious observer.

In Listing 4.1.3, we show how they can be concealed using a proxy object. To do so, we introduce the `cloak` keyword. Cloaking a code block indicates that the data within is classified according to a label. Therefore, values created within its *lexical* scope become faceted values within a proxy object. The outgoing information flows are then marked as being faceted so that the rest of the program may be rewritten much as they are with plain, uncloaked faceted values.

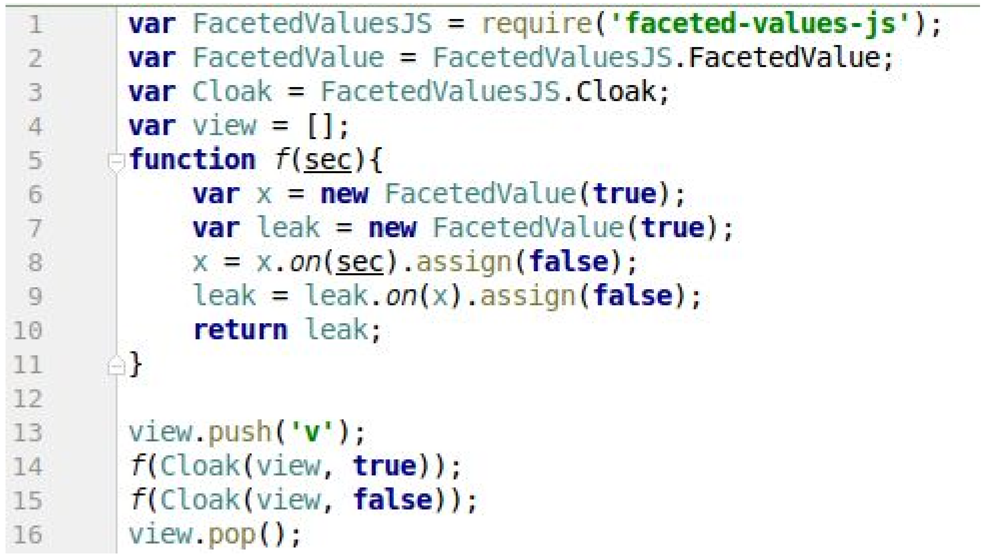


```
function f(sec){
    var x = true;
    var leak = true;
    if (sec)
        x = false;
    if (x)
        leak = false;
    return leak;
}

cloak v {
    f(true);
    f(false);
}
```

Listing 4.1.3: Cloaking-specific input source code.

Aside from merely making values faceted, they can be shielded from view using the `cloak` structure with the label `v`.



```
var FacetedValuesJS = require('faceted-values-js');
var FacetedValue = FacetedValuesJS.FacetedValue;
var Cloak = FacetedValuesJS.Cloak;
var view = [];
function f(sec){
    var x = new FacetedValue(true);
    var leak = new FacetedValue(true);
    x = x.on(sec).assign(false);
    leak = leak.on(x).assign(false);
    return leak;
}

view.push('v');
f(Cloak(view, true));
f(Cloak(view, false));
view.pop();
```

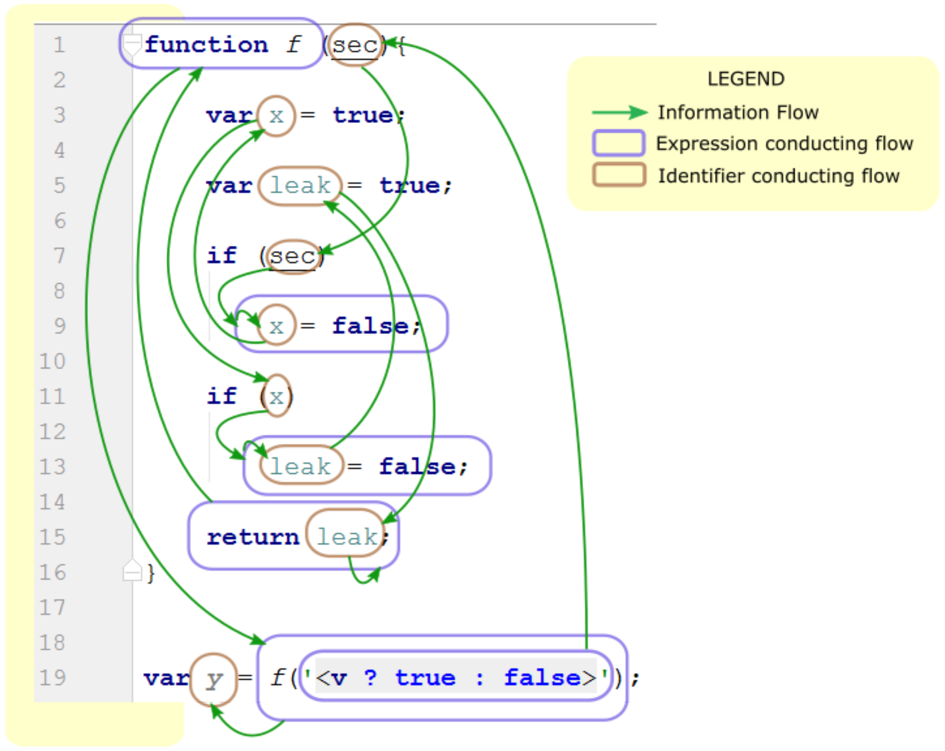
Listing 4.1.4: Cloaking-specific output source code.

Values within the cloak span of code become faceted values wrapped inside a proxy object with Cloak. Outside the field of view v, the proxy object returns the faceted value's public data. An observer needs not only a reference to the (cloaked) faceted value, but also needs to be within the correct view.

To make the source code transformations depicted in these Listings possible is the primary contribution of our project.

4.2: Graph of information flows overlaying abstract syntax tree

An unanticipated positive outcome of the experiment is the graphing of information flows during static analysis. It provides a novel approach to analyzing code. While engineers naturally think in terms of information flows to an extent, their understanding more commonly follows from the sequence of instructions and lexical flow of their program. Being built upon the abstract syntax tree, the understanding is born of the programming language's operational semantics.



```
function f (sec){

    var x = true;

    var leak = true;

    if (sec)

        x = false;

    if (x)

        leak = false;

    return leak;
}


var y = f('<v ? true : false>');
```

Figure 4.2.1: Information flows overlaying source code.

Not all flows are drawn — only those beginning with '`<v ? true : false>`'. It flows through the `CallExpression` `f` to the argument declaration `sec`, to the `Identifier` `sec`, to the body of the `if` statement's consequent `AssignmentExpression`, to the `Identifier` `x`, to the `VariableDeclarator` `x`, to the `Identifier` `x`, to the body of the if statement's consequent

`AssignmentExpression`, to the `Identifier leak`, to the `VariableDeclarator leak`, to the `Identifier leak`, to the `ReturnExpression`, to the `FunctionDeclaration f`, to the `CallExpression f`, and finally to the `VariableDeclarator y`.

Note that the flows do not necessarily follow instructions as a program counter would move through the program. Rather, they describe how the set of possible valuations of an expression in the program influences or is influenced

5: Conclusions

5.1: Functionality of faceted values

We were able to implement faceted values in plain JavaScript without native support. They can be used in place of unfaceted values, and have the desired properties laid out in the original, theoretical formulation of faceted values [6]. The implementation, along with copious amounts of JSDoc documentation, can be found in FacetedValue.js. The library's functioning is exhaustively tested, and its use cases near-exhaustively tested, in testFacetedValue.js. These tests can be run in Node.js with the NodeUnit package installed.

5.2: Usability of the library

The usability of the library, while certainly having room for improvement, demonstrates the ease with which faceted values may one day be used. Users are able to feed the library relatively simple code (as in Listing 4.1.1) that may not be functional, but does reflect the intended use of faceted values. It then transforms the source file into the complex code required to support faceted values (as in Listing 4.1.2). This suggests that a similar transformation can be done “under the hood” by a JavaScript runtime were it to provide native support.

5.3: (Absence of) Native support

Our design goals specified that our implementation must be done in standard JavaScript, without requiring any special constructs within the JavaScript runtime. We have done so, using pure unadulterated Node.js. This suggests that the logic by which we implemented faceted values can be translated into any given JavaScript compiler or interpreter in a (relatively) straightforward manner.

5.4: Syntactic sugar

One of our primary goals was to demonstrate ease of use for faceted values by giving them syntactic sugar. Our initial intent was to do so using the SweetJS library; this may well be true, or it may yet be false. SweetJS’s ability to rewrite source code on the lexical level is in excess of our needs, and a simpler module that allows to rewrite on the level of the abstract syntax tree is sufficient. To that end we selected and successfully used AST-Query in place of SweetJS. A side-benefit of this change was that the process of refactoring, and by extension our own library, has become much easier to debug and much more thoroughly documentable on the programming level.

5.5: Explicit information flow through the abstract syntax tree

An unanticipated outcome of this project, and its use of abstract syntax trees, is that as a side effect it produces graphs that clearly define how data flows through the program. It is a beautiful development, given that this project is a study in information flow security and analysis. How these graphs serve to complete our pipeline is outlined in Section 3.4.3. Some more tangential observations are made in Section 4.2.

It is my personal opinion that this was the most interesting outcome of the paper. Viewing software in the light of “information flow across the syntax tree” has applications outside topic of faceted values and can conceivably provide insights into any program’s functioning.

5.6: Static analysis reduces impact of faceted values

Initially we had believed that altering the behavior of a program to accommodate faceted values was a matter of dynamic analysis in which the program constantly examines values to see whether or not they are faceted and responds accordingly. If that were the case, the impact would depend on whether faceted values were given native support[14] or not.[15]

However, we were able to combine both dynamic and static analysis. The dynamic analysis is largely built-in to the methods in FacetedValue.js. The static analysis is performed in FacetedValuesJS.js, and is described in more detail in Section 3.4.3.

The upshot of this static analysis is that it determines which nodes in the abstract syntax tree can possibly conduct faceted values — by negation, we can determine which nodes cannot possibly conduct faceted values, and ignore them when refactoring.

The downshot is that we still cannot determine *when* nodes will possibly conduct faceted values according to the code path by which the faceted values are arriving. Were this the case, we could further refactor the code to separate incoming flows into separate nodes, and thus

[14] Given native support, this continual checking of values would be no worse than the same that is now being done for proxy objects.

[15] Not given native support, as in our program, this continual checking requires that every single operation be overwritten as a function that can take FacetedValue as input: a severe performance impact.

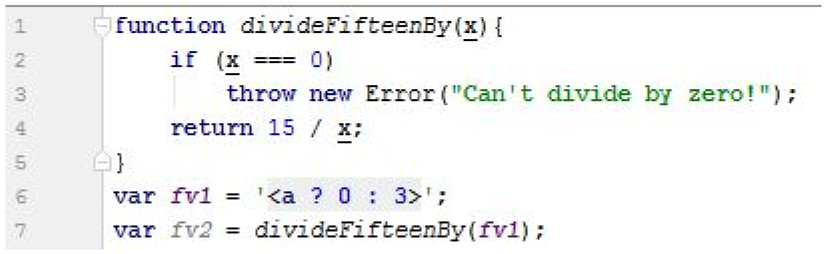
reduce the frequency with which faceted operations are unnecessarily performed on unfaceted values.

6: Future Work

There are many avenues for future work and exploration following this project, which are presented in no particular order:

6.1: Facet-specific terminating behaviors

It is unclear how we should handle behaviors and exceptions that terminate differently depending on the facet. One possibility is for the faceted value's operation methods to capture exceptional behavior, and then defer the reporting of that exceptional behavior to the time at which the value is retrieved through a proxy object's getter method.



```
function divideFifteenBy(x){
    if (x === 0)
        throw new Error("Can't divide by zero!");
    return 15 / x;
}
var fv1 = '<a ? 0 : 3>';
var fv2 = divideFifteenBy(fv1);
```

Listing 6.1.1: Facets with exceptions.

If the exception is thrown without being captured by the faceted value, any program that could catch it (or otherwise monitor the error report) would be able to deduce that the private value is zero.

6.2: Performance testing

Some amount of slowdown in processing is inevitable with the inclusion of faceted values, by the simple fact that any given value may be faceted and must be examined for the possibility. This slowdown is somewhat ameliorated by such checks already being made for proxy object values (of which faceted values can be a variation). Nonetheless, it is worth considering what the actual cost to performance is imposed by having faceted values, and how that cost may compare to competing solutions (such as secure multi-execution) in a variety of scenarios.

6.3: Further static (and dynamic!) analysis

Static analysis of the abstract tree allowed us to reduce the impact of implementing faceted values, as detailed in Section 5.6. There is more to be done on this front. In particular, deep analysis of the data that a faceted value's facets can contain may limit the potential reach of those facets, which in turn may reduce the parts of the abstract syntax tree that need to be refactored.

6.4: Information flow in programming languages

A particularly interesting and unanticipated outcome of this project was the graph of information flows overlaying the abstract syntax tree as described in Section 4.2. This provides a novel way of viewing, understanding, and debugging programs.

One possible future work would be to create a new programming language in which the information flow and lexical flow are closely matched. For such an endeavor, an important first step would be to examine how information and lexical flows compare in a variety of already-existing languages. A good starting list for this would be JavaScript, ANSI C, BASIC, LISP, Haskell, Ruby, and Scala.

Another future work that has great potential is to extend a debugger that allows developers to step through the code — not line-by-line or instruction-by-instruction, but instead — by following the information flow graph.

6.5: Exhaustive testing of the library

The library created for this project has some rudimentary tests completed, but by no means are they exhaustive. Given that the purpose of the program is to rewrite programs, and these programs can be of any conceivable size and make, the task of writing exhaustive tests is a project all its own. Nonetheless, it is a task that would need to be done before faceted values can be added to a real-world JavaScript runtime.

Most of the tests that need to be added are complete sample programs that can be fed into ASTquery_fiddling.js to put the library through its paces. To put it another way, we just need many, many programs to be rewritten using faceted values.

6.6: Scalability

A vulnerability of the method we use is that it only works with a single file at a time. In practice it may be necessary for it to span multiple source files. However, to do so invites memory issues as the code base it examines expands. It is possible that the program can be rewritten to write its processing phases to disk and save memory, producing incremental code transformations as a byproduct. For now, this is outside of this project’s scope, and would be a far-future consideration.

6.7: Eval support

The library does not currently support JavaScript's `eval` function. This is not essential, and it is questionable whether someone concerned enough with cybersecurity to use faceted values should be using `eval` anyway. However, a 2011 study suggests that a simple majority of the Internet's popular websites use eval, and a two-thirds majority is likely [4]. If we are to claim faceted values' support for general-purpose programming it behooves us to address eval. This would be a difficult task that requires imagination and ingenuity, and it is very likely that a solution for eval will prove computationally expensive. One possibility may be to recognize the use of eval, enclose it within a scope to give its contents a limited lifetime, and then use proxy objects to shuttle data to the eval'd code. Another possibility is to include a version of our faceted value pipeline at runtime that intercepts script loading and dynamically recompiles the eval'd code. In either case, it is worth considering the possibility of enhancing our pipeline to remove some or all of the eval'd code to begin with, as described by Jensen et al [5].

6.8: Attack detection

One of faceted values' shortcomings is that by design they do not prevent security breaches, instead concealing classified data and providing counterfeit data. We could use this to our advantage, as it opens opportunities to ensnare attackers with a confidence game. The first step in such a scheme would be to enhance faceted values with a mechanism that allows for attack detection. Potentially it could be a part of the Cloak proxy object. When an observer retrieves a counterfeit value through the proxy, it suggests that the observer is in some way illicit. The proxy object can then, in addition to producing the counterfeit value, record a stack trace with which the developer may identify and study attacks — or even introduce their own honeypots dynamically!

6.9: Environment-specific channel interdiction

As stated in our introduction, our library is not a freestanding solution. There is still the matter of relaying facets to observers. Who or what these observers are, and what channels exist to communicate with them, depend upon the environment. In NodeJS, the channels may consist of various system calls. In a web browser, the channels may be comprised of HTTP requests, changes to the Document Object Model, plugin requests, and more – all of which may change from one browser vendor or version to another. Our present assumption is that faceted values will only provide complete protection after a number of “companion” libraries have been created to provide interdiction on these channels.

7: References

1. Van Cutsem, T., and Miller, M. S. (2010, October). Proxies: design principles for robust object-oriented intercession APIs. In *ACM Sigplan Notices* (Vol. 45, No. 12, pp. 59-72). ACM.
2. Maffeis, S., Mitchell, J. C., and Taly, A. (2010, May). Object capabilities and isolation of untrusted web applications. In *Security and Privacy (SP), 2010 IEEE Symposium on* (pp. 125-140). IEEE.
3. Austin, T. H., Disney, T., and Flanagan, C. (2011, October). Virtual values for language extension. In *ACM SIGPLAN Notices* (Vol. 46, No. 10, pp. 921-938). ACM.
4. Richards, G., Hammer, C., Burg, B., and Vitek, J. (2011). The eval that men do. *ECOOP 2011–Object-Oriented Programming*, 52-78. In Proc. 25th European Conference on Object-Oriented Programming, volume 6813 of LNCS. Springer, July 2011.
5. Jensen, S. H., Jonsson, P. A., and Møller, A. (2012, July). Remedying the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*(pp. 34-44). ACM.
6. Austin, T. H., and Flanagan, C. (2012, January). Multiple facets for dynamic information flow. In *ACM Sigplan Notices* (Vol. 47, No. 1, pp. 165-178). ACM.
7. Van Cutsem, T., and Miller, M. S. (2013, July). Trustworthy proxies. In *European Conference on Object-Oriented Programming* (pp. 154-178). Springer, Berlin, Heidelberg.
8. Disney, T., Faubion, N., Herman, D., and Flanagan, C. (2014, October). Sweeten your JavaScript: Hygienic macros for ES5. In *ACM SIGPLAN Notices* (Vol. 50, No. 2, pp. 35-44). ACM.
9. Austin, T., Knowles, K., and Flanagan, C. (2014). *Typed faceted values for secure information flow in haskell*. Technical Report UCSC-SOE-14-07, University of California, Santa Cruz.
10. Rafnsson, W., and Sabelfeld, A. (2016). Secure multi-execution: Fine-grained, declassification-aware, and transparent. *Journal of Computer Security*, *24*(1), 39-90.
11. ECMA International. ECMAScript 2016 Language Specification, 7th Edition.
12. Kannan, P., Austin, T.H., Stamp, M., Disney, T., and Flanagan, C. (2016). *Virtual Values for Taint and Information Flow Analysis.* Workshop on Meta-Programming Techniques and Reflection (META). ACM.
13. Austin, T. H., Schmitz, T., and Flanagan, C. (2017). Multiple Facets for Dynamic Information Flow with Exceptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *39*(3), 10.