CS 252 Project Report: Faceted Values in Haskell

Andrew Kalenda*[1]

[1]Department of Computer Science, San José State University

Faceted values are a cyber-security technique that provide different views of data depending on the observer, ensuring privacy where needed and protecting data from mischievous third-party programs. Currently, more compelling examples of this technique are needed; to this end, a Haskell faceted values library has been created. The intent of this project is to give a concrete implementation of the library and demonstrate faceted values' efficacy in a strongly, statically typed language.

1 Introduction

Information flow controls[4] are cyber-security mechanisms by which in-the-wild programs may confine classified data to safe channels so that sensitive information cannot leak. (In particular, cannot leak to third-party programs.) Historically, information flow control has been a consideration of the individual software developer, but tracking the flow of information is (especially in the untamed wilds of web development) prohibitively difficult and tedious. Even a mindful developer is hard-pressed to allocate precious time and energy when other concerns are mounting. Thus the birth of programmatic controls that will relieve the burden.

Secure multi-execution[5–7] is one such control mechanism. It splits program execution into two paths: *high* and *low*. A datum's life begins on the *low* path. On *low*, data may be written to any output, public or private. However, when a datum becomes determined by classified information, it is permanently elevated to the *high* path. This elevation can either occur as the datum is determined directly (an *explicit flow* of information) or indirectly (an *implicit flow* of information). The *high* execution path has its output restricted to authorized channels, thus preserving the sanctity of data.

Another mechanism, *faceted values*[2], simulates secure multi-execution in a single process. A faceted value is a monad containing a private and public value - the facets - which express respective views of a datum depending on the observer. To an unprivileged accessor, the true (private) value is obscured; thus, *high* and *low* execution paths are simulated and unnecessary.

This report intends to expose difficulties an end-user may have with faceted values and the library when they are not familiar with the particulars of information flow, secure multi-execution, and faceted values. It also attempts to explain the groundwork for faceted values with a target audience of upper-division undergraduate students. References are available for more technical and theoretical details.

Please note that pseudocode contained herein is not properly Javascript (or any other language), and only bears a passing resemblance.

2 Information Flow Controls

Information flow[4] is just as the words describe: how information flows from one data structure or process to another. By way of analogy, you could imagine a computer program as

*andrew.kalenda@sjsu.edu

a complex system of pipes, pumps, and machines through which flow a variety of potentially dangerous fluids (water, oil, steam, gasoline, chemicals, etc). The architect of this system needs to ensure that these flows do not taint each other in unintended ways, and that there are no problematic leaks. This is done through *information flow controls* (which could be likened to valves, channels, and meters in the pipe analogy). Consider the following pseudocode:

```
var b = user.getBalance()
var isRich = b > 999999
if (isRich)
   out = 'Youre a millionaire!'
else
   out = 'Youre not a millionaire!'
save(out)
print(out)
```

There is a clear vulnerability here: if malicious code can be inserted into the same scope as this code block, it can read the value of `b` at any time to find the user's balance. So a conscientious programmer may change it to the following:

```
function getOutput() {
   var b = user.getBalance()
   var isRich = b > 999999
   if (isRich)
      return 'Youre a millionaire!'
   else
      return 'Youre not a millionaire!'
}
out = getOutput()
save(out)
print(out)
```

The scoping of the function conceals the value of `b`, preventing it from leaking. This is good, because the user's account balance is obviously private information. However, a more subtle leak remains here. The user's balance determines the value of `b`, which in turn determines `isRich`, which in turn determines the function's return value, which in turn determines the value of `out`. Thus, there is an *implicit* flow of information from `b` to `out`, and `out` contains private information.

Malicious code could still infer whether or not the user is a millionaire. Our program needs to compensate by returning a different value when viewed by an unintended audience:

```
function saveAndPrintOutput() {
   if (executionSecured)
      print("Invalid user")
      return
   if (user.getBalance() > 999999)
      var out = 'Youre a millionaire!'
   else
      var out = 'Youre not a millionaire!'
   save(out)
   print(out)
}
```

This approach is enough for many purposes, such as in this code sample. However there are scenarios wherein cutting an execution short under certain conditions can allow an observer to

make inferences on hidden data based on how long the code took to execute! For example, this contrived piece of code:

```
function foo() {
   if (user.getBalance() > 999999)
      return hiddenProcess()
   else
      return hiddenProcessThatTakesTwiceAsLong()
}
```

Even should malicious code be found in the same scope as this function, no values whatsoever are exposed. Despite that, this code could then be executed and the time it takes to do so observed. If it runs quickly, then the malware can infer that the user is in fact a millionaire! This can be prevented by executing both processes:

```
function foo() {
   var v1 = hiddenProcess()
   var v2 = hiddenProcessThatTakesTwiceAsLong()
   return (user.getBalance() > 999999) ? v1 : v2
}
```

3 Secure multi-execution

Any or all of the above approaches continue to have a problem, though. They rely on the programmer being continually mindful of vulnerabilities and staying up-to-date on current best practices, beyond what is covered here and in the context of their own application:

> "The Dark Arts [...] are many, varied, ever-changing, and eternal. Fighting them is like fighting a many-headed monster, which, each time a neck is severed, sprouts a head even fiercer and cleverer than before. You are fighting that which is unfixed, mutating, indestructible. [...] Your defenses [...] must therefore be as flexible and inventive as the arts you seek to undo." [8]

This is difficult even for dedicated information security experts. When a programmer is focused on other fields, it becomes prohibitively difficult. It would be wonderful if the secure data simply did not exist for the unauthorized observer, and that is part of what *secure multi-execution*[5–7] achieves.

In secure multi-execution, there are two execution paths: *hi* and *lo*. When data is determined by private data it is jumped to the hi-execution channels (which can see all data, but can only write to authorized channels). When an unauthorized channel attempts to get private data, it receives dummy values instead. The following is an inexact analogy for how it might operate:

```
function getBalance() {
   hi.write(balance);
   lo.write(0);
}
```

Both the hi- and lo-executions automatically run in parallel with one another. This ensures that an outside observer is unable to make inferences on private data based on how long she or he must wait for the program to execute, because it is the same length of time regardless of which channel they actually access.

4 Faceted values

Secure multi-execution necessitates enveloping entire programs in hi and lo paths, even for portions where implicit data flows are not actually a problem. Furthermore, it is still up to the programmer to keep track of private data and ensure that it safely within the hi paths. It would be ideal if selection of the hi and lo paths were tied directly to the private data. This is the purpose of *faceted values*, as described in more detail in Austin and Flanagan's *Multiple Facets for Dynamic Information Flow*[2].

First, for illustrative purposes, let's imagine the previous code snippet rewritten as follows:

```
function getBalance() {
   return (Environment.isHi()) ? balance : 0;
}
hi.executeAndReturn(getBalance)
lo.executeAndReturn(getBalance)
```

Into this, we introduce the concept of the *faceted value*: a simple object that contains a public and private value, with a label to mediate which of the two facets may (in the future) be seen by an observer. It is written as `<label ? private : public>`:

```
function getBalance() {
   return <isHi ? balance : 0>;
}
<isHi ? hi : lo>.executeAndReturn(getBalance)
```

Despite its similarity to the ternary operator, it is important to note that the faceted value is not one or the other facet. Rather, like Schrodinger's cat, it is simultaneously both possibilities. The returned value is both `balance` **and** zero, only appearing to be one or the other depending on the observer at the moment of observation. Even as observation ends, it remains both the public and private value.

Similarly, the faceted value `<isHi ? hi : lo>` is simultaneously both the hi- and lo-execution path of secure multi-execution. It is, at all times when dealing with faceted values, implied that we have these multiple executions as appropriate in whatever programming environment we're using, and so the above would simply be written as the following pseudocode:

```
function getBalance() {
   return <isHi ? balance : 0>
}
```

Faceted values are monadic, algebraic data types. We can combine and composite them in a variety of ways. For example:

```
var a1 = <Alice ? 2 : 3>
var a2 = <Alice ? 5 : 7>
a1 * a2 == <Alice ? 10 : 21> // True!

var b1 = <Bob ? 5 : 7>
a1 * b1 == <Alice ? <Bob ? 10 : 14> : <Bob ? 15 : 21>> // True!

<Alice ? 0 : 0> == 0 // True
<Alice ? <Alice ? 42 : 0> : 0> == <Alice ? 42 : 0> // True!
<Alice ? <Bob  ? 42 : 0> : 0> == <Bob ? 42 : 0> // False!

/*
```

```
 * When entering a code branch that is decided by a faceted value,
 * all values creates therein are themselves implicitly faceted.
 * If this were a powerpoint presentation, now would be a great
 * time for an "images of angels become angels" Dr. Who slide!
 */
var d = <Don ? 222 : 11>
var e = "merrrp"
if (d > 50)
   e = "Pish"
   e == <Don ? "Pish" : "merrrp"> // True
else
   e = "Posh"
   e == <Don ? "merrrp" : "Posh"> // True
e == <Don ? <Don ? "Pish" : "merrrp"> : <Don ? "merrrp" : "Posh">> // True

/*
 * To truly simulate secure multiexecution, we best treat code
 * branches as facets themselves, and execute both. This may not
 * be strictly necessary for a given scenario, but it gives the
 * strongest gaurantees. Given the equivalencies of faceted values
 * as an algebraic data type, we can simplify the above:
 */
e == <Don ? <Don ? "Pish" : "merrrp"> : <Don ? "merrrp" : "Posh">> // True
e == <Don ? "Pish" : <Don ? "merrrp" : "Posh">> // True
e == <Don ? "Pish" : "Posh"> // True
```

Of course, implementing these truths in a living programming language is quite the task.

5 *Haskell-Faceted* Library

As of the writing of this paper, the Haskell-Faceted library has been created and described [3]. This paper goes on to describe my own experiences with it as a neophyte. The central piece of the library is, of course, the data type itself for faceted values:

```
{-
   pseudocode ===> Haskell code :
   <Alice ? 42 : 9> ==> Faceted "Alice" (Raw 42) (Raw 9)
-}
type Label = String
data Faceted
   = Raw a
   | Faceted Label (Faceted a) (Faceted a)
```

It is also defined as a functor, applicative functor, and monad. The details of these implementations are important only insofar as they work to make faceted values work as one would expect, given the requirements we have described for faceted values as an algebraic data type in the previous section. They allow us to do something like the following:

```
facetedSum = (+) <$> facetedA <*> facetedB
-- supposed to be equivalent, but appears to be bugged:
facetedSum = do
   facetedC <- facetedA + facetedB
   return facetedC
```

By the same token, we could implement pseudocode from the previous section:

```
doStuff = do
   a1 <- Faceted "Alice" (Raw 2) (Raw 3)
   a2 <- Faceted "Alice" (Raw 5) (Raw 7)
   print $ (*) <$> a1 <*> a2
   b1 <- Faceted "Bob" (Raw 5) (Raw 7)
   print $ (*) <$> a1 <*> b1
```

More complex is the implementation for branching. Here I am forging ahead into territory not yet fully explored; the following should be considered an advisory on where to start, not definitive statements!

```
{- psuedocode:
    var d = <Don ? 222 : 11>
    var e = "merrrp"
    if (d > 50)
       e = "Pish"
    else
       e = "Posh"
-}
foo4 :: PC -> Faceted String
foo4 activeLabels = do
   let d = Faceted "Don" (Raw 222) (Raw 11)
      e = Raw "merrrp"
      condF = (>) <$> Raw 50 <*> d
      thenF = Raw "Pish"
      elseF = Raw "Posh"
   facetedIf condF thenF elseF e

facetedIf :: Faceted a -> Faceted b -> Faceted b -> Faceted b -> Faceted b
facetedIf condFacet thenFacet elseFacet placebo = mergeBrnch
   where trueBranch = commuteFacets condFacet thenFacet placebo
      falsBranch = commuteFacets condFacet placebo thenFacet
      mergeBrnch = commuteFacets condFacet trueBranch falsBranch

commuteFacets :: Faceted a -> Faceted b -> Faceted b -> Faceted b
commuteFacets (Faceted layble _ _) = Faceted layble
```

Note the addition of the facetedIf, and its accompanying commuteFacets. This is an ungraceful way of moving the facets of the conditional statement onto the branches which depend on it. Nonetheless, it does the job. Some other additions that may prove useful:

```
    -- We can create sub-FIO's, and use this within those sub-FIOs to depend
    -- a faceted value on whatever currently active labels there are
    makeMultifaceted :: PC -> a -> a-> Faceted a
    makeMultifaceted pc seekrit placebo = pcF pc (Raw seekrit) (Raw placebo)

    -- When we get whatever value the observer can see, we also get what must
    -- what must and must not be visible in order to get the result.
    project2 :: View -> Faceted a -> (PC, a)
    project2 ____ (Raw v) = ([], v)
    project2 view (Faceted k prv pub)
       | k `elem` view = (Public k : accumPrv, vPrv)
```

```
      | otherwise    = (Private k : accumPub, vPub)
      where (accumPrv, vPrv) = project2 view prv
            (accumPub, vPub) = project2 view pub

-- If the public and private facets are the same, we can collapse them
-- This compacts horizontally, we should also have one that compacts vertically,
-- and combine the two into a single recursive function that is invoked
-- whenever we expect it will be useful (i.e. facetedIf)
compact :: (Eq a) => Faceted a -> Faceted a
compact (Raw v) = Raw v
compact (Faceted lab prv pub)
   | prv == pub = compact pub
   | otherwise = Faceted lab (compact prv) (compact pub)
```

6 Conclusion

It's the end of the semester and this project (and by extension this very paper) is unfinished, as was expected. Hopefully if nothing else it will prove invaluable in showing how the library can be improved for usability issues, potentially missing functionality, and perhaps even elucidating some of the details on how faceted values work versus how they are intended to work. In particular, usability would be greatly improved if the following two snippets worked as expected:

```
sum1 facetedA facetedB = do
   facetedC <- facetedA + facetedB
   return facetedC
sum2 facetedA = do
   facetedC <- facetedA + 42
   return facetedC
```

While not mentioned elsewhere in this report, the Hackage packaging for Faceted is unreliable. The project should continue in some form to put the library entirely through its paces, as there may be other functions missing.

References

[1] Austin, T. H., & Flanagan, C. (2009). *Efficient purely-dynamic information flow analysis.* ACM Sigplan Notices, 44(8), 20-31.

[2] Austin, T. H., & Flanagan, C. (2012). *Multiple facets for dynamic information flow.* ACM SIGPLAN Notices, 47(1), 165-178.

[3] Austin T.H., Knowles K. & Flanagan C. (2014). *Typed Faceted Values for Secure Information Flow in Haskell.* ???[1].

[4] Denning, D. E., & Denning, P. J. (1977). *Certification of programs for secure information flow.* Communications of the ACM, 20(7), 504-513.

[5] Devriese, D., & Piessens, F. (2010, May). *Noninterference through secure multi-execution.* In Security and Privacy (SP), 2010 IEEE Symposium on (pp. 109-124). IEEE.

[1] TODO: Find out what to put in here!

[6] Jaskelioff, M., & Russo, A. (2012). *Secure multi-execution in haskell.* In Perspectives of Systems Informatics (pp. 170-178). Springer Berlin Heidelberg.

[7] Rafnsson, W., & Sabelfeld, A. (2013, June). *Secure multi-execution: fine-grained, declassification-aware, and transparent.* In Computer Security Foundations Symposium (CSF), 2013 IEEE 26th (pp. 33-48). IEEE.

[8] Rowling, J. K. (2013). *Harry Potter and the half-blood prince (Vol. 6).* Bloomsbury Publishing.