# Implementation of IF in Cooma



## Implementation

This document contains implementation information about adapting Cooma Expression Safety Proof to the current Cooma language.

### 1. Syntax

Currently in the Cooma parser, all types are expressions. To extend this to support the `!` character, we add an additional `SecT` construct above the other types in `Cooma.syntax`:

```
Expression =
  ...
  | Expression '!'                                          {SecT, none,
4}
  ...
```

We then rely on checks in `SemanticAnalyser.scala` to ensure that only certain expressions may be used in this place. As per the proof these are the following *base types*:

- Int
- Boolean
- String
- Unit

We also support the following built-in capabilities:

- Reader
- ReaderWriter

- Writer

The enforcement of these expressions is located in `SemanticAnalyser.scala` and performed by the following function:

```
def checkSecretType(e : Expression) : Messages =
    e match {
        case IntT() | StrT() | BoolT() | UniT() | ReaderT() | WriterT() |
ReaderWriterT() => noMessages
        case _ => error(e, s"cannot have a secret ${show(e)}")
    }

Types and capabilities implemented in the symbol table must also have
analgous 'secret' versions which are used when a trailing `!` is detected.
The main changes are that anywhere base types occur in the records or
variants are now wrapped in a `SecT` node to indicate a secret type.

```scala
// Secret boolean
val secBoolT : Expression =
    VarT(Vector(FieldType("False", SecT(UniT())), FieldType("True",
SecT(UniT()))))

// Secret capabilities
val secReaderT : Expression =
    RecT(Vector(
        FieldType("read", FunT(ArgumentTypes(Vector()), SecT(StrT())))
    ))

val secReaderWriterT : Expression =
    RecT(Vector(
        FieldType("read", FunT(ArgumentTypes(Vector()), SecT(StrT()))),
        FieldType("write",
FunT(ArgumentTypes(Vector(ArgumentType(Some(IdnDef("s")), SecT(StrT())))),
SecT(UniT())))
    ))

val secWriterT : Expression =
    RecT(Vector(
        FieldType("write",
FunT(ArgumentTypes(Vector(ArgumentType(Some(IdnDef("s")), SecT(StrT())))),
SecT(UniT())))
    ))
```
```

**NOTE**: Unlike previous version of this document, we do not support secret records, variants, types or functions. We calculate the security levels for these types based on the base types used in them.

## 2. Semantic Analysis

This section is used to collect various implementataion notes on `SemanticAnalyser.scala`.

### 2.1 Subtype

We can add our subtype realtion, which enforces our information flow, to the `subtype` function. Since secret types ares wrappers over normal types, we can easily identify when we are checking some type `t` against a secret type `u` and pull the underlying type `s` out for comparison.

```
def subtype(t : Expression, u : Expression) : Boolean =
  ...
  case (_, SecT(s)) =>
    subtype(t, s)
  ...
```

### 2.2. Aliasing

We need to add corresponding `alias` and `unalias` cases to link our analagous implementations in `SymbolTable.scala` to the alias secret boolean and capabilities. The alias of the secrets are the same as there public counterparts, prefixed with 'sec':

```
def alias(e : Expression) : Expression =
  ...
  case `secBoolT           => SecT(BoolT())
  case `secReaderT`        => SecT(ReaderT())
  case `secReaderWriterT` => SecT(ReaderWriterT())
  case `secWriterT`        => SecT(WriterT())
  ...
```

The corresponding cases added to `unalias`:

```
def unalias(n : ASTNode, t : Expression) : Option[Expression] =
  ...
  case SecT(BoolT())          => Some(secBoolT)
  case SecT(ReaderT())        => Some(secReaderT)
  case SecT(ReaderWriterT()) => Some(secReaderWriterT)
  case SecT(WriterT())        => Some(secWriterT)
  ...
```

### 2.3. Security Property Implementation

The security computation functions from the like proof above are implemented as such:

```
def secProp(e : Expression) : Boolean =
    tipe(e) match {
        case Some(TypT()) => e match {
            case FunT(ArgumentTypes(as), r) =>
                as.forall(a => secLevel(a.expression) <= secLevel(r) &&
                    secProp(r)
            case _ => true
        }
        case _ => true
    }

def secLevel(e : Expression) : Int =
    tipe(e) match {
        case Some(TypeT()) => e match {
            case FunT(_, t1) => secLevel(t1)
            case RecT(fs)    => rowSec(fs)
            case VarT(fs)    => rowSec(fs)
            case _ : SecT    => 1
            case _           => 0
        }
        case _ => 0
    }

def rowSec(fs : Vector[FieldType]) : Int =
    fs.foldLeft(0)((l : Int, f : FieldType) => math.max(l,
secLevel(f.expression)))
```

The `secProp` function is responsible for generating a proposition in which all the necessary relationships between types are laid out. If the `secProp` doesn't hold, due to one of the relationships not being true, then we know we have a leak and thus and error is produced.

The secProp function first check if the given expression is a `TypT`. This is because the function is only defined on types; our security realtion only deals with types. In other cases where we don't have a type we simply return `true`. In cases where we do have a type, we are only concerned with enforcing a certain relationship within function types since other types don't have relationships that affect the security property. If we do find a function type, we make sure that each argument type has a security level of lesser or equal value to the return type. This ensures that the return type of a function is at least as secure as the most secure type input into the funciton.

The `secLevel` function computes an integer value for a given type. For base types, if it's secret it's given a value of 1, otherwise it must be public and is given a value of 0. For functions, we compute the security level based on the return type of the function. For records and variants, we take the maximum value in the row as the level for the whole record or variant. This is done using the `rowSec` function.

These functions are used to enforce the security property holds for the Cooma language. Whilst the proof shows the chech performed on almost all constructs in the language, we only need implement it for two constructs: functions (and by extension defs) and matches.

This arises from the difference between theory and implementation. In the proof checks need to be performed on applications and other constructs to ensure that the underlying types also abide by the security property. For example, we know that on the left of an application there must be a function at some point (even if there is a long chain of functions returning functions) and thus if the function does not abide by the security property, it will eventually be picked up when the anlaysis on the AST gets to that node. Similarly for identifiers, if types don't abide by the security property, they will be picked up in other places in the tree so it's unneccesary to check for them at the given node.

#### 2.3.1. Function Checks

The function checks are implemented as:

```
def checkFuntion(f : Fun) : Message =
    tipe(f) match {
        case Some(fun @ FunT(_, _)) =>
            secProp(fun) match {
                case true  => noMessages
                case false =>
                    error(f, "security property violated, return type is
less secure then one or more of the arguments")
            }
        case _ => noMessages
    }
```

The function first checks that the type of the argument can be typed as a function. In the case where it can't, we have stumbled across an ill-typed function and can rely on other checks in `SemanticAnalyser.scala` to pick that up an report the actual error; we are only interested in asserting the security peroprty for well-typed functions. If we do have a well-typed function then we check the corresponding type-signature in relation to the security property. If it's true, the function has the property, if not, we report an error.

**NOTE**: It should be noted that the error reporting is quite limited at this time. It will **not** tell you the offending arguments, only report that the function violates the security property.

#### 2.3.2. Match Checks

The match checks are implemented as:

```
def checkMatchSec(e : Expression, cs : Vector[Case]) : Messages =
    cs.foldLeft(noMessages)((m, c) => (tipe(e), tipe(c.expression)) match
{
        case (Some(t), Some(u)) =>
            secProp(FunT(ArgumentTypes(Vector(ArgumentType(None, t))), u))
```

```
match {
                case true  => m
                case false =>
                    m ++ error(c.expression, "security property violated,
case return value is less secure then a field in the variant being matched
on")
            }
        case _ => m
    }
```

The match check works similarly to the function check except that it has to perform the check on each case in the match expression. We can see that we start with no errors and only add an error if the security property for a particular case is violated. Similarly, we also don't report errors on ill-typed cases for the same reason as functions. Unlike functions, our error reporting can pin-point which case it was that violates the property. It's likely however, that either all cases will violate the property or none, since cases are enforced to have the same return type.

A slight difference here is that we actually construct a function signature for each case and perform the security property check on that. This is due to the fact that matches are represented slightly differently in the proof. In the proof a macth is actually types as a special function, but in Cooma that are typed as the return type of the case (provided all the match expression is well-typed).

**NOTE**: Previously, additional checks for the security property were put in place in the `tipe` function so that functions and matches that violated the property would return `None`. These have been removed in the current version to significantly reduce the number of changes to the code base.

### 2.3. Primitive Operations

**NOT YET IMPLEMENTED**

We also need to consider the additional changes that we need to make to primitive operations. Our model dictates that we should be able to perform primitive operations between secret and public data as long as the result is also secret i.e. `Int! + Int = Int!`. This would mean special expections need to be made for primitive operations when dealing with an operand of type `SecT`.

Currently, Cooma handles prmitive operations as such:

- A primitive AST node is check by the `checkPrimitive` function in `SemanticAnalyser.scala`.
- This function looks up the primitive in `SymbolTable.scala` by using the identifier.
- If the identifier is found it then performs a check to make sure the given number of arguments match the number of expected number of arguments.

The `primitveTypesTable` is a mapping from the various operations in `Ints`, `Strings` and the `Equal` operation to function types `FunT`. We need to expand this to deal with secret types, since the function types will be different i.e. they should all return secret types.

## 3. Current List of Changes

- Add `!` expression case to `Cooma.syntax`.
- Add secret boolean and built-in capability definitions to `SymbolTable.scala`.
- Add secret boolean and built-in capability cases to `alias` function in `SemanticAnalyser.scala`.
- Add secret boolean and built-in capability cases to `unalias` function in `SemanticAnalyser.scala`.
- Add `secProp`, `secLevel` and `rowSec` functions to `SemanticAnalyser.scala` for computation of security levels.
- Add `checkFunction` to check functions for security property in `SemanticAnalyser.scala`.
- Add `checkMatchSec` to check matches for security property in `SemanticAnalyser.scala`.
- Add `checkSecretType` to ensure only certain expression can be secret in `SemanticAnalyser.scala`.
- Add case to `tipe` for `SecT` in `SemanticAnalyser.scala`
- Add case to `subtype` for `SecT` in `SemanticAnalyser.scala`
- Add test file `InformationFlowTests.scala`