# TDA342
# Security - Part II

QUFEI WANG
JACOB J. NILSSON

March 21, 2020

## Task 1

First, we use the same definition for data type *Sensitivity* as we did in Part I.

```
data Sensitivity = Secret | Public deriving (Show, Eq)

instance Monoid Sensitivity where
  mempty = Public
  mappend Public Public = Public
  mappend _ _ = Secret

instance Semigroup Sensitivity where
  (<>) = mappend
```

Second, we add the same value constructor *Sec Expr* to the data type *Expr* as we did in Part I to adjust to the keyword *secret*.

```
data Expr = Lit Integer
          | Expr :+: Expr
          | Var Name
          | Let Name Expr Expr
          | NewRef Expr
          | Deref Expr
          | Expr := Expr
          | Sec Expr -- ^ new value constructor
          | Catch Expr Expr
  deriving (Show)
```

Then we modify the definition of data types *Env*, *Store* and functions *lookupVar*, *localScope*, *newRef*, *deref* to keep track of labeled values in the environment. We list the changes in the code as follows:

```
-- | An environment maps variables to labeled values.
type Env = Map Name (Labeled Value)

-- | We need to keep track of the store containing the
    labeled values of
-- our references. We also remember the next unused
    pointer, furthermore,
-- we keep track of the stacks of 'try catch' expressions
     to prevent
-- implicit security leakage
data Store = Store { nextPtr  :: Ptr
                   , heap     :: Map Ptr (Labeled Value)
                   , tryStack :: [Sensitivity]
                   }

-- | Environment manipulation
lookupVar :: Name -> Eval (Labeled Value)
lookupVar x = do
  env <- ask
  case Map.lookup x env of
    Nothing -> throwError (UnboundVariable x)
    Just v  -> return v

-- | update variable binding
localScope :: Name -> Labeled Value -> Eval a -> Eval a
localScope n v = local (Map.insert n v)

-- | Create a new reference containing the given labeled
    value.
newRef :: Labeled Value -> Eval Ptr
newRef v = do
              store <- get
              let ptr      = nextPtr store
                  ptr'     = 1 + ptr
                  ts       = tryStack store
                  newHeap  = Map.insert ptr v (heap store
                      )
              put (Store ptr' newHeap ts)
              return ptr

-- | Get the labeled value of a reference. Crashes with
    our own
```

```
-- "segfault" if given a non-existing pointer.
deref :: Labeled Ptr -> Eval (Labeled Value)
deref (LV sp p) = do st <- get
                     let h = heap st
                     case Map.lookup p h of
                       Nothing -> throwError
                           SegmentationFault
                       Just (LV sv v)  -> return (LV (sp
                           `mappend` sv) v)

-- | Updating the labeled value of a reference. Has no
    effect if the
-- reference doesn't exist.
(=:) :: MonadState Store m => Ptr -> Labeled Value -> m (
    Labeled Value)
p =: v = do store <- get
            let heap' = Map.adjust (const v) p (heap
                store)
            put (store {heap = heap'})
            return v
```

Finally, we present the code which evaluates an *Expr* to a labeled value, it also contains the functionality for task 2 and task 3.

```
-- | evaluate an expression
eval :: Expr -> Eval (Labeled Value)
eval (Lit n)       = return (LV Public n)
eval (a :+: b)     = do
  LV s1 v1 <- eval a
  LV s2 v2 <- eval b
  return $ LV (s1 `mappend` s2) (v1 + v2)
eval (Var x)       = lookupVar x
eval (Let n e1 e2) = do LV s v <- eval e1
                        localScope n (LV s v) (eval e2)
eval (Sec e)       = do LV _ v <- eval e
                        return $ LV Secret v
eval (NewRef e)    = do LV s v <- eval e
                        ptr <- newRef (LV s v)
                        return $ LV s ptr
eval (Deref e)     = do LV s p <- eval e
                        peepAndUpdate s
                        deref (LV s p)
eval (pe := ve)    = do LV sp p <- eval pe
                        LV sv v <- eval ve
                        mg <- peepGuard
```

```
                        if sp == Public && (sv == Secret
                            || mg == Just Secret)
                          then fail "security violation"
                          else p =: LV (sp `mappend` sv)
                               v
eval (Catch e1 e2) = do
  pushGuard Public
  a <- catchError (eval e1) (\err -> f e2)
  popGuard
  return a
  where f e = do
          LV s v <- eval e2
          mg <- peepGuard
          case mg of
            Nothing -> return (LV s v)
            Just s' -> return (LV (s `mappend` s') v)
```

## Task 2

Our definition of function *runWithInput* is as follows:

```
runWithInput :: Eval a -> IO (Either Err a)
runWithInput x = do
  putStr "Enter the value of secret variable input: "
  str <- getLine
  let p  = read str :: Ptr
      x' = localScope "input" (LV Secret p) x
  return $ runEval x'
```

It basically interprets the input from the user as a value of type *Ptr*, binds this value to the reserved variable *input* with a *Secret* label, and uses the updated environment to evaluate the whole expression.

## Task 3

We added error handling to our interpreter with *ExceptT* on the outside instead of inside, because the order matters in our latter attempt to keep track of implict flows, for a reason which we will explain later.

```
newtype Eval a = MkEval (ExceptT Err (StateT Store (
    ReaderT Env Identity)) a)
```

```
  deriving (Functor, Applicative,
            Monad, MonadState Store, MonadReader Env,
                MonadError Err)

runEval :: Eval a -> Either Err a
runEval (MkEval err) = runIdentity $ runReaderT
                       (evalStateT
                           (runExceptT err)
                        emptyStore) emptyEnv
```

In order to deal with implicit flows, we see each '*try*...*catch*' statement as occurring in its own level of stack which is labeled by a security guard, *Secret* or *Public*. In such a case, whenever a *Public* reference is to be written by a *Secret* value or the write operation happens under a *Secret* guard, a security exception is raised. The guard is always set to *Public* when first entering into a *try*...*catch* statement. Any reference to a *Secret* value or pointer in an evaluation of a dereference expression (which is $Deref$, and we only consider the exceptions that might be incurred by this kind of evaluation as being influential) updates the security guard to *Secret*, thus any following calculation in the same scope is deemed to be dependent on some secret data.

The functions dealing with the stack of security guards as monad computations are listed as follows, for the definition of the security stacks in the data type *Store* under the attribute *tryStack*, and the computational parts dealing with the implicit flows in function *eval*, please see the code posted above or refer to the source file.

```
-- | push a security gurad for a new 'try catch'
    calculation
pushGuard :: Sensitivity -> Eval ()
pushGuard s = modify $ \store ->
  let np = nextPtr store
      hp = heap store
      ts = tryStack store
  in Store np hp (s:ts)

-- | update the current security gurad
updateGuard :: Sensitivity -> Eval ()
updateGuard g = modify $ \store ->
  let np = nextPtr store
      hp = heap store
      ts = tryStack store
  in Store np hp (g:(tail ts))

-- | pop a security guard when leaving a 'try catch'
    calculation
```

```
popGuard :: Eval ()
popGuard = modify $ \store ->
  let np = nextPtr store
      hp = heap store
      ts = tryStack store
  in Store np hp (tail ts)

-- | check if current security guard exists
peepGuard :: Eval (Maybe Sensitivity)
peepGuard = do
  s <- get
  let ts = tryStack s
  case ts of
    [] -> return Nothing
    _  -> return . Just $ head ts
```

**Questions: What is the difference between implementing your secure interpreter with *ExceptT* on the inside and on the outside?**

With *ExceptT* being implemented inside, the changes made on the *Store* state will be reverted for the exception handler if an exception occurs during an evaluation of an expression. This, however, will not happen if *ExceptT* is implemented outside.

This is because the combined result of applying *ExceptT* outside and *StateT* inside on an inner monad $m\ a$ is turning it to $m\ (Either\ e\ a, s)$, while the other way is to turn it to $m\ (Either\ e\ (a, s))$. For the purpose of our program, because we need to keep the changes on the state to determine whether the following operation is secure or not, we choose the outside version of implementation.