Hybrid Proofs Domain Specific Language

Matthew Hotchkiss, Nes Cohen

March 16, 2022

1 Project Overview

The goal of this project was to create a simple DSL, *hybrids*, to be used for hybrid proofs in cryptography. *hybrids* acts as a proof assistant. The project is written in Haskell, a functional programming language that allows for easy typing. The types constructed in the project can be used to construct entire hybrid proof sequences, such as the one discussed in the 'Deconstruction of a Proof' section. It also includes some additional elements, such as converting libraries to Latex and even performing simple encryptions, such as for OTP. By no means is this project exhaustive or perfect, but it includes all the overarching types necessary to construct proofs, which could be expanded into a larger-scale production - or simply used by other cryptography enthusiasts who want to use it in their own crypto projects. The project is hosted on GitHub, and is published to Hackage as well for cryptographic devotees and big-project-thinkers alike.

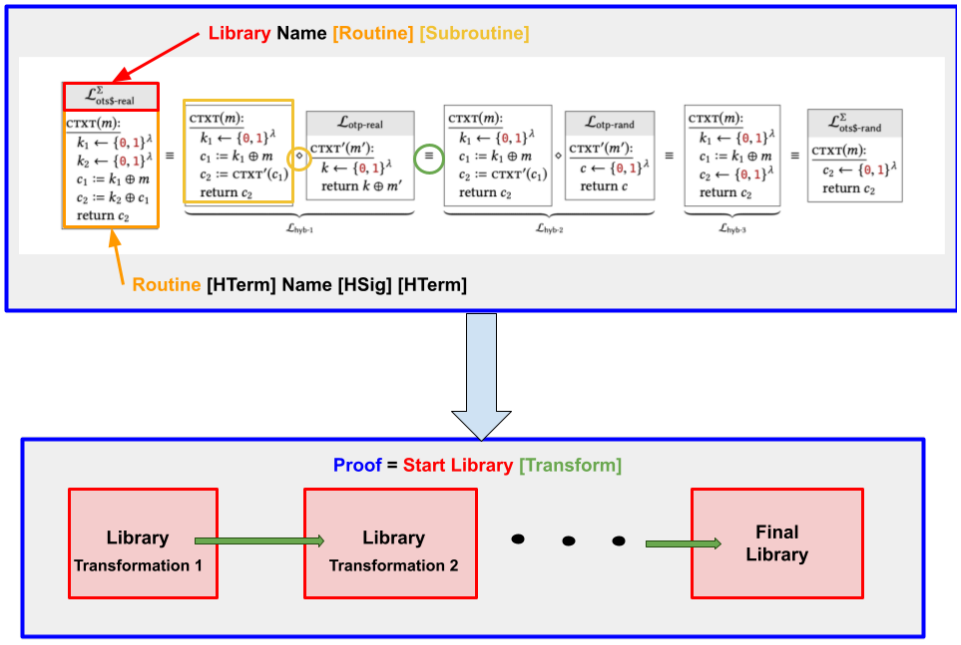
Repository: `https://github.com/nescohen/hybrids`

2 Deconstruction of a Proof

In order to represent a hybrid proof relevant to cryptography via a domain specific language (DSL), the constituents of the visual proof should be understood individually as forms of data. In other words, the proof must be decomposed into components such that the DSL can interpret the proof in parts (such as libraries, routines, etc.). Consider the diagram (at the end of the section) to express the decomposition of a proof.

A proof is a sequence of libraries, linked together by a sequence of changes and relationships. In other words, a hybrid proof can be expressed in terms of its starting library, along with the sequence of changes and relationships between it and its final library. These changes (transformations) could include in-lining subroutines, for example. Keep in mind, a transformation also stores the relationship, so indistinguishability or interchangeability are represented between libraries in the sequence of changes.

How is an individual library represented? Well, a library must have a name (otp-real, prg, etc.), followed by any routines within the main body of the library, and any subroutines linked to the library. The routines themselves are no different from each other in data type but simply in relation to the library, so we can represent routine and subroutine all the same.



A routine is similarly identified by its name (CTXT, EAVESDROP, etc.). A routine also might have an initialization routine, such as key generation that occurs outside of the routine call but within the library. Otherwise, a routine accepts arguments (HSig) and has a body (HTerms) that will execute.

3 Abstract Syntax Tree

The basic building block of the *hybrids* language is the expression. All values in *hybrids* are bitstrings. Expressions are built from a combination of literals, variables, exclusive-or expressions, append expressions, and function calls. The Haskell datatype "HExpr" represents the set of possible expressions.

```
data HExpr where
    Variable    :: HVar -> HExpr
    Literal     :: BitString -> HExpr
    Xor         :: HExpr -> HExpr -> HExpr
    Append      :: HExpr -> HExpr -> HExpr
    Call        :: HName -> HArgs -> HExpr
```

In order to represent complete functions are programs, *hyrbids* defines terms (HTerm)". The terms can represent variable assignment, random sampling, control flow, function returns, and composition of statements. The Haskell datatype "HTerm" represents the set of possible terms.

```
data HTerm where
    Assign   :: HVar -> HExpr -> HTerm
    Gets     :: BitWidth -> HVar -> HTerm
    (:>)     :: HTerm -> HTerm -> HTerm
    Return   :: HExpr -> HTerm
    Loop     :: Bound -> Bound -> HTerm -> HTerm
    If       :: Boolean -> HTerm -> HTerm
```

Finally, *hybrids* supports functions (Routine), scopes (Block) and libraries (Library). These correspond with the matching concepts commonly used in hybrid proofs. This is useful because a hybrid proof is typically concerned with showing that a particular library is interchangeable with or indistinguishable from another library. Here are the haskell definitions:

```
data Routine where
    Rout :: Maybe HTerm -> HName -> HSig -> HTerm -> Routine

newtype Block = Block [Routine]

data Library where
    Lib :: HName -> Block -> Maybe Block -> Library
```

4 Transformations

In order to show proofs in *hybrids*, we use AST transformations to represent an assertion that two HTerms are interchangeable/indistinguishable.

```
newtype Interchangeable = Inter (HTerm -> HTerm)

newtype Indistinguishable = Indis (HTerm -> HTerm)

data Transform = InterT Interchangeable
               | IndisT Indistinguishable
```

The set of valid ASTs is closed under valid transformations, therefore transformations may be composed.

```
composeTransform :: Transform -> Transform -> Transform
```

The full source of `composeTransform` and other functions in this section is included in the appendix.

The *hybrids* package also includes built-in valid transformations, such as "inline". Inline takes a function name as its only argument. It replaces each call expression of the named function with a modified version of the function's body. Then it removes the named function.

```
inline :: HName -> Interchangeable
```

hybrids allows the user to define their own their own general transformations or to simply assert the equality of two libraries. The goal of the built-in transformations is to provide a sound proof assistant, but then allow for the user to provide additional definitions to add more flexibility. When defining custom transformations, the responsibility is now on the user to ensure their transformations and assertions are sound.

5 Evaluation

In addition to being used to construct proofs, *hybrids* supports evaluation of Programs. This may be used for illustrative purposes or even to run a proven library unchanged in production.

```
evalHTerm :: HContext -> HTerm -> Either Error (HContext, Maybe
    BitString)
```

The `HContext` argument allows for variables to be bound and the caller to provide a secure source of random bits.

6 Usage

One-time secrecy is a common security definition representing an encryption scheme suitable to send a single message securely. Here is the *hybrids* representation of one-time secrecy (real):

```
otsReal :: BitWidth -> Library
otsReal size =
    Lib "ots-real" (Block [Rout Nothing "EAVESDROP" (HSig ["m"
    ])
        (
            Gets size (HVar "k")
            :> Assign (HVar "c") (Xor (Variable $ HVar "k") (
    Variable $ HVar "m"))
            :> Return (Variable $ HVar "c")
        )
    ]) Nothing
```

LaTeXtranslation:

```
libToLatex (otsReal bits8)
```

```
\[
\titlecodebox{$\lib{ots-real}^\Sigma$}{
\underline{$\subname{EAVESDROP}(m)$:} \\
\> $k \gets \bits^{8}$\\
\> $c := k \oplus m$\\
\> return $c$
}
\]
```

Rendered:

$\mathcal{L}^{\Sigma}_{\text{ots-real}}$

EAVESDROP(m):

$k \leftarrow \{0,1\}^8$

$c := k \oplus m$

return c

One-time secrecy (rand):

```
otsRand :: BitWidth -> Library
otsRand size =
    Lib "ots-rand" (Block [Rout Nothing "EAVESDROP" (HSig ["m"
    ])
         (
             Gets size (HVar "c")
             :> Return (Variable $ HVar "c")
         )
    ]) Nothing
```

LaTeXtranslation:

```
libToLatex (otsRand bits8)
```

```
\[
\titlecodebox{$\lib{ots-rand}^\Sigma$}{
\underline{$\subname{EAVESDROP}(m)$:} \\
\> $c \gets \bits^{8}$\\
\> return $c$
}
\]
```

Rendered:

$\mathcal{L}^{\Sigma}_{\text{ots-real}}$

EAVESDROP(m):

$k \leftarrow \{0,1\}^8$
$c := k \oplus m$
return c

7 Appendix

All code with cabal build files is available in the github repository:
`https://github.com/nescohen/hybrids`
The *hybrids* code is open source, provided under the MIT license.

7.1 Transformation Composition

```
composeTransform :: Transform -> Transform -> Transform
composeTransform (InterT (Inter t1)) (InterT (Inter t2)) =
    InterT (Inter (t1 . t2))
```

```
composeTransform t1                  t2                  =
    IndisT (Indis ((unwrapT t1) . (unwrapT t2)))

unwrapT :: Transform -> HTerm -> HTerm
unwrapT (InterT (Inter f)) = f
unwrapT (IndisT (Indis f)) = f
```

7.2 Inline Transformation

```
-- inline: transformation replaces all instances of named
-- function with function contents, then removes function.
inline :: HName -> Interchangeable
inline fName = Inter trans
  where
    trans term | Just routine <- extractRoutine fName term =
        fromMaybe term $ replace routine term
               | otherwise = term

    containsE :: HExpr -> Maybe HArgs
    containsE (Variable _)     = Nothing
    containsE (Literal _)      = Nothing
    containsE (Xor e1 e2)
        | Just args <- containsE e1 = Just args
        | otherwise                 = containsE e2
    containsE (Append e1 e2)
        | Just args <- containsE e1 = Just args
        | otherwise                 = containsE e2
    containsE (Call name args) = if name == fName
                                 then Just args
                                 else Nothing

    replaceE :: HTerm -> HExpr -> HExpr
    replaceE body (Xor e1 e2)         = Xor (replaceE body e1)
    (replaceE body e2)
    replaceE body (Append e1 e2)      = Append (replaceE body
    e1) (replaceE body e2)
    replaceE _    c@(Call name _)
        | name == fName = Variable (HVar $ map toLower fName)
        | otherwise     = c
    replaceE _    e                   = e

    rewriteCtx :: HSig -> HArgs -> Map HName HExpr
    rewriteCtx (HSig [])     (HArgs [])     = M.empty
    rewriteCtx (HSig (v:vs)) (HArgs (e:es)) = M.insert v e $
    rewriteCtx (HSig vs) (HArgs es)
    rewriteCtx _             _              = error "Call
    argument mismatch during inline rewrite"
```

```
rewriteBody ctx ((:>) t1 t2) = rewriteBody ctx t1 :>
rewriteBody ctx t2
rewriteBody ctx (Return e)   = Assign (HVar $ map toLower
fName) $ rwrtBodyE ctx e
rewriteBody ctx (Assign v e) = Assign v $ rwrtBodyE ctx e
rewriteBody _   body         = body

rwrtBodyE :: Map HName HExpr -> HExpr -> HExpr
rwrtBodyE _   e@(Literal _)            = e
rwrtBodyE ctx v@(Variable (HVar name)) = case M.lookup name
 ctx of
                                           Just e -> e
                                           Nothing -> v
rwrtBodyE ctx (Xor e1 e2)              = Xor (rwrtBodyE ctx
 e1) (rwrtBodyE ctx e2)
rwrtBodyE ctx (Append e1 e2)           = Append (rwrtBodyE
ctx e1) (rwrtBodyE ctx e2)
rwrtBodyE ctx (Call name (HArgs es))   = Call name (HArgs (
map (rwrtBodyE ctx) es))

replace :: (HSig, HTerm) -> HTerm -> Maybe HTerm
replace (sig, body) ((:>) t1 t2)  =
    case replace (sig, body) t1 of
        Nothing  -> replace (sig, body) t2
        Just t1' -> case replace (sig, body) t2 of
            Nothing  -> Just t1'
            Just t2' -> Just (t1' :> t2')
replace (sig, body) term@(Assign v e)
    | Just args <- containsE e = Just (rewriteBody (
rewriteCtx sig args) body
                                          :> Assign v (
replaceE body e))
    | otherwise                = Just term
replace (sig, body) term@(Return e)
    | Just args <- containsE e = Just (rewriteBody (
rewriteCtx sig args) body
                                          :> Return (replaceE
 body e))
    | otherwise                = Just term
replace _    f@(Routine n _ _) = if n == fName then Nothing
 else Just f
replace _    hterm             = Just hterm
```

7.3 Eval HTerm

```
evalHTerm :: HContext -> HTerm -> Either Error (HContext, Maybe
    BitString)
evalHTerm ctx (Assign (HVar name) e) = do
```

```
    bs <- evalHExpr ctx e
    return (hVarDef name bs ctx, Nothing)
evalHTerm ctx (Gets (BitWidth n) var) = do
    let (bs, g') = randomBytesGenerate ((n `div` 8) + 1) (
    ctxRand ctx)
        bString = BitString $ take n $ bytesToBits bs
    (ctx', _) <- evalHTerm ctx (Assign var (Literal bString))
    return (ctx' { ctxRand = g' }, Nothing)
evalHTerm ctx ((:>) t1 t2) = do
    (ctx', res) <- evalHTerm ctx t1
    case res of
        Nothing -> evalHTerm ctx' t2
        Just r  -> return (ctx', Just r)
evalHTerm ctx (Return e) = do
    bs <- evalHExpr ctx e
    pure (ctx, Just bs)
evalHTerm ctx (Routine name sig body) =
    return (hFnDef name sig body ctx, Nothing)
```

7.4 LaTeX Translation

```
libToLatex :: Library -> IO ()
libToLatex (Lib name block _blk) =
    putStrLn ( "\\[\n"
            ++ "\\titlecodebox{$\\lib{" ++ unpack (replace (
    pack "$") (pack "\\$") (pack name)) ++ "}^\\Sigma$}{\n"
            ++ blockToLatex block ++ "\n}"
            ++ "\n\\]")

blockToLatex :: Block -> String
blockToLatex (Block []) = ""
blockToLatex (Block (x:xs)) = routToLatex x ++ blockToLatex (
    Block xs)

subsToLatex :: Block -> String
subsToLatex (Block []) = ""
subsToLatex (Block (x:xs)) = subToLatex x ++ subsToLatex (Block
     xs)

subToLatex :: Routine -> String
subToLatex rout = "\\link\n"
                ++ "\\hlcodebox{\n"
                ++ routToLatex rout
                ++ "}"

routToLatex :: Routine -> String
routToLatex (Rout (Just initial) name sigs body) =
    bodyToLatex 0 initial ++ "\\\\\n"
```

```
 ++ "\\underline{$\\subname{" ++ name ++ "}(" ++ show sigs ++ "
    )$:} \\\\\n"
 ++ bodyToLatex 1 body
routToLatex (Rout Nothing name sigs body) =
    "\\underline{$\\subname{" ++ name ++ "}(" ++ show sigs ++ "
    )$:} \\\\\n"
 ++ bodyToLatex 1 body


bodyToLatex :: Int -> HTerm -> String
bodyToLatex lvl (Assign v e)  =
  (duplicateStr "\\>" lvl) ++ " $" ++ show v ++ " := " ++
    exprToLatex e ++ "$"
bodyToLatex lvl (Gets bits v) =
  (duplicateStr "\\>" lvl) ++ " $" ++ show v ++ " \\gets " ++
    bitsToLatex bits ++ "$"
bodyToLatex lvl ((:>) t1 t2)  =
  bodyToLatex lvl t1 ++ "\\\\\n" ++ bodyToLatex lvl t2
bodyToLatex lvl (Return e)    =
  (duplicateStr "\\>" lvl) ++ " return $" ++ exprToLatex e ++ "
    $"
bodyToLatex lvl (Loop b1 b2 t) =
  (duplicateStr "\\>" lvl) ++ " for $i=" ++ show b1 ++ "$ to $"
     ++ show b2 ++ "$:\\\\\n" ++ bodyToLatex (lvl+1) t
bodyToLatex lvl (If b t) =
  (duplicateStr "\\>" lvl) ++ " if (" ++ boolToLatex b ++ ")
    :\\\\\n" ++ bodyToLatex (lvl+1) t
bodyToLatex _   _        = error "Routine print not defined"

duplicateStr :: String -> Int -> String
duplicateStr _ 0 = ""
duplicateStr str n = str ++ duplicateStr str (n-1)

exprToLatex :: HExpr -> String
exprToLatex (Variable (HVar name)) = name
exprToLatex (Literal bs) = ppBits (toBits bs)
exprToLatex (Xor e1 e2) = exprToLatex e1 ++ " \\oplus " ++
    exprToLatex e2
exprToLatex (Append e1 e2) = exprToLatex e1 ++ " \\| " ++
    exprToLatex e2
exprToLatex c = show c

bitsToLatex :: BitWidth -> String
bitsToLatex (BitWidth n) = "\\bits^" ++ "{" ++ show n ++ "}"

boolToLatex :: Boolean -> String
boolToLatex (Eq expr1 expr2) = exprToLatex expr1 ++ " == " ++
    exprToLatex expr2
boolToLatex (Gt expr1 expr2) = exprToLatex expr1 ++ " > " ++
```

```
    exprToLatex expr2
boolToLatex (Lt expr1 expr2) = exprToLatex expr1 ++ " < " ++
    exprToLatex expr2
boolToLatex (Neq expr1 expr2) = exprToLatex expr1 ++ " \\neq "
    ++ exprToLatex expr2
boolToLatex (Gte expr1 expr2) = exprToLatex expr1 ++ " \\geq "
    ++ exprToLatex expr2
boolToLatex (Lte expr1 expr2) = exprToLatex expr1 ++ " \\leq "
    ++ exprToLatex expr2
boolToLatex (Empty expr) = exprToLatex expr ++ " empty"
boolToLatex (Undef expr) = exprToLatex expr ++ " undefined"
```