Authenticated Data Structures, Generically, in Haskell

Haskell eXchange 2018

Dr. Lars Brünjes, IOHK

2018-10-11

Agenda

- What are Authenticated data structures?
- Example: Merkle trees.
- A simple ad-hoc ADS.
- ADS's generically.
- Authenticated lists demo.

Authenticated data structures

An *authenticated datastructure (ADS)* is a data structure whose operations can be carried out by an untrusted prover, the results of which a verifier can efficiently check as authentic.

Andrew Miller et al.

Rough idea

How can this possibly work?

Rough idea

How can this possibly work?

The rough idea is to use (cryptographic) hashing: The verifier just needs hashe(s) of the datastructure(s), the prover includes preimages to those hashes in its proofs.

Cryptographic Hash functions

A (cryptographically secure) *hash function* is a function that takes arbitrary bitstrings to bitstrings of a fixed length with the following additional properties:

Cryptographic Hash functions

A (cryptographically secure) *hash function* is a function that takes arbitrary bitstrings to bitstrings of a fixed length with the following additional properties:

- collision resistant

collision resistant

We are very unlikely to find two inputs which give the same output, no matter how hard we try.

Cryptographic Hash functions

A (cryptographically secure) *hash function* is a function that takes arbitrary bitstrings to bitstrings of a fixed length with the following additional properties:

- collision resistant
- hiding

hiding

Given a hash, it is infeasible to find its associated input, and the optimal way to do so is to try every possibility uniformly randomly.

Hashing in Haskell

In Haskell, we can use the excellent cryptonite library for hashing. For these slides, we'll use MD5, but any hashing algorithm would do.

```haskell
data Hash = ...
```

```haskell
hash :: Binary a => a -> Hash
```

```
GHCi> hash "Haskell"
e74f20fc19d925fafccacc7ab837249e
```

```
GHCi> hash (42 :: Int)
7e0535868cd45dff74884bfba0fa1594
```

Merkle trees

In Bitcoin, Merkle trees are used to enable Simple Payment Verification (SPV) nodes.

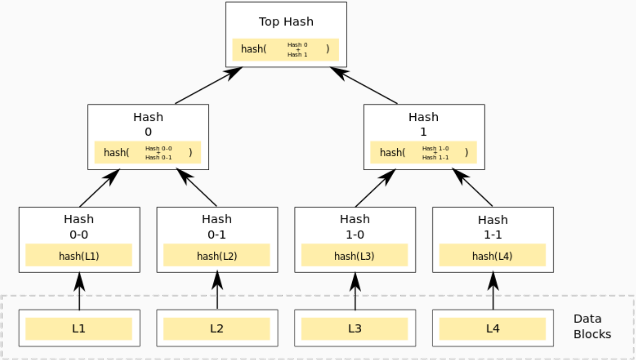


Figure 1: Example Merkle tree *(Wikipedia)*

A simple tree type

Instead of Merkle trees, we will consider a very similar type as our running example:
Let’ define a type for simple binary trees with data in the leaves...

```
data Tree a = T a | N (Tree a) (Tree a)
  deriving (Show, Generic, Binary)
```

A simple tree type

Instead of Merkle trees, we will consider a very similar type as our running example:
Let’ define a type for simple binary trees with data in the leaves...

```
data Tree a = T a | N (Tree a) (Tree a)
  deriving (Show, Generic, Binary)
```

...and types representing paths in such trees:

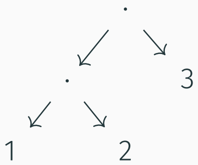
```
data Direction = L | R deriving Show
type Path = [Direction]
```

Tree lookup

Following a `Path`, we can `lookup` the value at the corresponding leaf (ignoring partiality for simplicity's sake):

```
lookup :: Path -> Tree a -> a
lookup []       (T a)   = a
lookup (d : ds) (N l r) =
  lookup ds $ case d of L -> l; R -> r
```

```
GHCi> let t = N (N (T 1) (T 2)) (T 3)
GHCi> lookup [L, R] t
2
```



Tree lookup with proving & verifying

We want to to split `lookup` between a prover and a verifier.

The prover holds the tree, the verifier only knows the tree's (modified) `hash'`:

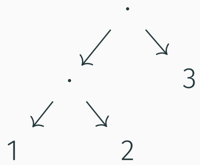
```
hash' :: Binary a => Tree a -> Hash
hash' (T a)   = hash a
hash' (N l r) = hash (hash' l, hash' r)
```

Proving tree lookup

```
prove :: Binary a
      => Path -> Tree a -> (a, [(Hash, Hash)])
prove []       (T a)   = (a, [])
prove (d : ds) (N l r) =
  let (a, hs) = prove ds $ case d of L -> l; R -> r
  in  (a, (hash' l, hash' r) : hs)
```

```
GHCi> prove [L, R] t
```

```
(2, [ ( c6f318528e5de4532ec597d3be978c8e
      , e4151c03023facf27bb46dd21c5bf6fd)
    , ( 69503798ddf28ee3fa2358a5ab9def30
      , 3fd723e4cfdb39c30f6352495b3c023f)
    ])
```



Verifying tree lookup

```
verify  :: Binary a
        => Path -> Hash -> (a, [(Hash, Hash)])
        -> Bool
verify []       h (a, [])            = h == hash a
verify (d : xs) h (a, (l, r) : hs) =
  h == hash (l, r)
    && verify xs (case d of L -> l; R -> r) (a, hs)
verify _        _ _                  = False
```

```
GHCi> verify [L, R] (hash' t) $ prove [L, R] t
True
```

```
GHCi> verify [L, R] (hash' t) $ prove [L, L] t
False
```

What we gain

- The *verifier* only needs the `hash'` of the `Tree`, not the `Tree` itself (note that the proof size is logarithmic in the tree size!)

What we gain

- The *verifier* only needs the `hash'` of the `Tree`, not the `Tree` itself (note that the proof size is logarithmic in the tree size!)
- In order to cheat, the *prover* would have to create a hash collision.

Disadvantages

- We had to implement more or less the same algorithm twice, once for the prover, once for the verifier.

Disadvantages

- We had to implement more or less the same algorithm twice, once for the prover, once for the verifier.
- Functions `prove` and `verify` have to be carefully designed for this to work.

Disadvantages

- We had to implement more or less the same algorithm twice, once for the prover, once for the verifier.
- Functions `prove` and `verify` have to be carefully designed for this to work.
- We had to come of with the custom hash function `hash'`.

Disadvantages

- We had to implement more or less the same algorithm twice, once for the prover, once for the verifier.
- Functions `prove` and `verify` have to be carefully designed for this to work.
- We had to come of with the custom hash function `hash'`.
- If we want to use a data structure other than `Tree` or want to support more operations that just `lookup`, we have to think and work hard and do a new proof of correctness.

Authenticated Datastructures, Generically

In *Authenticated Datastructures, Generically*, Andrew Miller, Michael Hicks, Jonathan Katz, Elaine Shi, describe a generic way to construct authenticated datastructures in OCaml.

Authenticated Datastructures, Generically

In *Authenticated Datastructures, Generically*, Andrew Miller, Michael Hicks, Jonathan Katz, Elaine Shi, describe a generic way to construct authenticated datastructures in OCaml.

- Extend the type system by adding a new type λa for each OCaml type a.

Authenticated Datastructures, Generically

In *Authenticated Datastructures, Generically*, Andrew Miller, Michael Hicks, Jonathan Katz, Elaine Shi, describe a generic way to construct authenticated datastructures in OCaml.

- Extend the type system by adding a new type λa for each OCaml type a.
- Formally add functions `auth` $: a \to \lambda a$ and `unauth` $: \lambda a \to a$ which are *inverse* to each other.

Authenticated Datastructures, Generically

In *Authenticated Datastructures, Generically*, Andrew Miller, Michael Hicks, Jonathan Katz, Elaine Shi, describe a generic way to construct authenticated datastructures in OCaml.

- Extend the type system by adding a new type λa for each OCaml type a.
- Formally add functions `auth` : $a \to \lambda a$ and `unauth` : $\lambda a \to a$ which are *inverse* to each other.
- Change the compiler, so that `auth` and `unauth` are treated differently for prover and verifier.

Authenticated Datastructures, Generically

In *Authenticated Datastructures, Generically*, Andrew Miller, Michael Hicks, Jonathan Katz, Elaine Shi, describe a generic way to construct authenticated datastructures in OCaml.

- Extend the type system by adding a new type λa for each OCaml type a.
- Formally add functions `auth` $: a \to \lambda a$ and `unauth` $: \lambda a \to a$ which are *inverse* to each other.
- Change the compiler, so that `auth` and `unauth` are treated differently for prover and verifier.

Instead of modifying GHC, we will instead use a free monad to achieve a similar effect!

Authenticated Datastructures, Generically (cntd.)

Prover

Verifier

$$\begin{tikzcd} a \arrow[d, bend right, "\texttt{auth}"'] \\ \lambda a \arrow[u, bend right, "\texttt{unauth}"'] \end{tikzcd}$$

Authenticated Datastructures, Generically (cntd.)

Prover

- $\lambda a \sim a$.

$$
\begin{tikzcd}
a \arrow[d, bend right, "\texttt{auth}"'] \\
\lambda a \arrow[u, bend right, "\texttt{unauth}"']
\end{tikzcd}
$$

Verifier

- $\lambda a \sim$ `Hash`.

Authenticated Datastructures, Generically (cntd.)

Prover

- $\lambda a \sim a$.
- `auth` a does nothing.

$$\begin{tikzcd} a \arrow[d, bend right, "\texttt{auth}"'] \\ \lambda a \arrow[u, bend right, "\texttt{unauth}"'] \end{tikzcd}$$

Verifier

- $\lambda a \sim$ `Hash`.
- `auth` a hashes a.

Authenticated Datastructures, Generically (cntd.)

Prover

- $\lambda a \sim a$.
- `auth` a does nothing.
- `unauth` x does nothing to get its result, but writes x to the proof-stream.

$$\begin{tikzcd} a \arrow[d, bend right, "\texttt{auth}"'] \\ \lambda a \arrow[u, bend right, "\texttt{unauth}"'] \end{tikzcd}$$

Verifier

- $\lambda a \sim$ `Hash`.
- `auth` a hashes a.
- `unauth` h reads a from the proof-stream and checks that `hash a == h`.

Auth

```haskell
data Auth a = P a | V Hash deriving Show
```

```haskell
toHash :: Binary a => Auth a -> Hash
toHash (P a) = hash a
toHash (V h) = h
```

```haskell
instance Binary a => Binary (Auth a) where
  put = put . toHash
  get = V <$> get
```

Auth

```haskell
data Auth a = P a | V Hash deriving Show
```

```haskell
toHash :: Binary a => Auth a -> Hash
toHash (P a) = hash a
toHash (V h) = h
```

```haskell
instance Binary a => Binary (Auth a) where
  put = put . toHash
  get = V <$> get
```

Crucially, when we serialize an `Auth a`, we “truncate” it to its hash, so proofs will be “short”.

Reminder: Free monads

```haskell
data Free f a =
    Pure a
  | Free (f (Free f a))
  deriving Functor
```

```haskell
instance Functor f => Applicative (Free f) where
  pure = return
  (<*>) = ap
```

```haskell
instance Functor f => Monad (Free f) where

  return = Pure

  Pure a >>= cont = cont a
  Free f >>= cont = Free $ (>>= cont) <$> f
```

AuthF & AuthM

```haskell
data AuthF a where
  A :: Binary b => b -> (Auth b -> a) -> AuthF a
  U :: Binary b => Auth b -> (b -> a) -> AuthF a
```

```haskell
deriving instance Functor AuthF
```

```haskell
type AuthM a = Free AuthF a
```

```haskell
auth :: Binary a => a -> AuthM (Auth a)
auth a = Free $ A a Pure
```

```haskell
unauth :: Binary a => Auth a -> AuthM a
unauth x = Free $ U x Pure
```

Authenticated trees

Using `Auth`, we slightly modify our `Tree` type and `lookup`:

```
data Tree a
   = T a | N (Auth (Tree a)) (Auth (Tree a))
  deriving (Show, Generic, Binary)
```

```
lookup :: Binary a
       => Path -> Auth (Tree a) -> AuthM a
lookup p x = do
  t <- unauth x
  case (p, t) of
    ([]   , T a)   -> return a
    (d : ds, N l r) -> lookup ds $
      case d of L -> l; R -> r
```

Interpretation for the prover

```
runProver  :: MonadWriter Builder m
           => AuthM a -> m a
runProver (Pure a)                = return a
runProver (Free (A a cont))       =
  runProver $ cont $ P a
runProver (Free (U (P a) cont)) = do
  tell $ lazyByteString $ encode a
  runProver $ cont a
```

```
runProver' :: AuthM a -> (a, ByteString)
runProver' m =
  let (a, b) = runWriter $ runProver m
  in (a, toLazyByteString b)
```

Interpretation for the verifier

Verification can fail, so let's define an appropriate error type:

```
data AuthError =
    SerError String
  | HashMismatch
  deriving Show
```

Interpretation for the verifier (cntd.)

```haskell
runVer :: ( MonadReader ByteString m
          , MonadError AuthError m)
       => AuthM a -> m a
runVer (Pure a)              = return a
runVer (Free (A a c))        = runVer $ c $ V $ hash a
runVer (Free (U (V h) c)) = do
  bs <- ask
  case decodeOrFail bs of
    Left (_, _, e)  -> throwError $ SerError e
    Right (bs', _, a)
      | hash a == h -> local (const bs') $ runVer $ c a
      | otherwise  -> throwError HashMismatch
```

```haskell
runVer' :: AuthM a -> ByteString -> Either AuthError a
runVer' m bs = runExcept $ runReaderT (runVer m) bs
```

Revisiting our example

Let's recover our example tree in this setting!

```
t, t' :: Auth (Tree Int)
t = fst $ runProver' $ do
  t1  <- auth $ T 1
  t2  <- auth $ T 2
  t12 <- auth $ N t1 t2
  t3  <- auth $ T 3
  auth $ N t12 t3
t' = V (toHash t)
```

```
GHCi> t
P (N (P (N (P (T 1)) (P (T 2)))) (P (T 3)))
GHCi> t'
V b5bd6ae28129b46d66d4f20924aa24ef
```

Revisiting our example (cntd.)

```
GHCi> let proof =
  snd $ runProver' [L, R] $ lookup t
GHCi> runVer' (lookup [L, R] t') proof
Right 2
GHCi> runVer' (lookup [L, L] t') proof
Left HashMismatch
```

`AuthT`

More generally, instead of using `Free` to define our monad `AuthM`, we can use `FreeT` to define a monad transformer `AuthT`:

```
newtype AuthT m a = AuthT (FreeT AuthF m a)
```

Another example: authenticated lists (demo)

As another simple example, we can define authenticated lists…

```haskell
data AList a = Nil | Cons a (Auth (AList a))
  deriving (Show, Generic, Binary)
```

…and use them to implement a simple stack API:

```haskell
push :: Binary a
     => a
     -> Auth (AList a)
     -> AuthM (Auth (AList a))
push a x = auth $ Cons a x
```

Another example: authenticated lists (demo)

As another simple example, we can define authenticated lists...

```
data AList a = Nil | Cons a (Auth (AList a))
  deriving (Show, Generic, Binary)
```

...and use them to implement a simple stack API:

```
pop :: Binary a
    => Auth (AList a)
    -> AuthM (Maybe a, Auth (AList a))
pop l = do
  xs <- unauth l
  return $ case xs of
    Nil       -> (Nothing, l)
    Cons a l' -> (Just a, l')
```

Thank you for your attention!

- EMail: `lars.bruenjes@iohk.io`
- Twitter: `@LarsBrunjes`
- GitHub: `https://github.com/brunjlar/generic-auth`