NYU

Logical Relations for Formally Verified

Authenticated Data Structures

Simon Oddershede Gregersen

joint work with Chaitanya Agarwal and Joseph Tassarotti

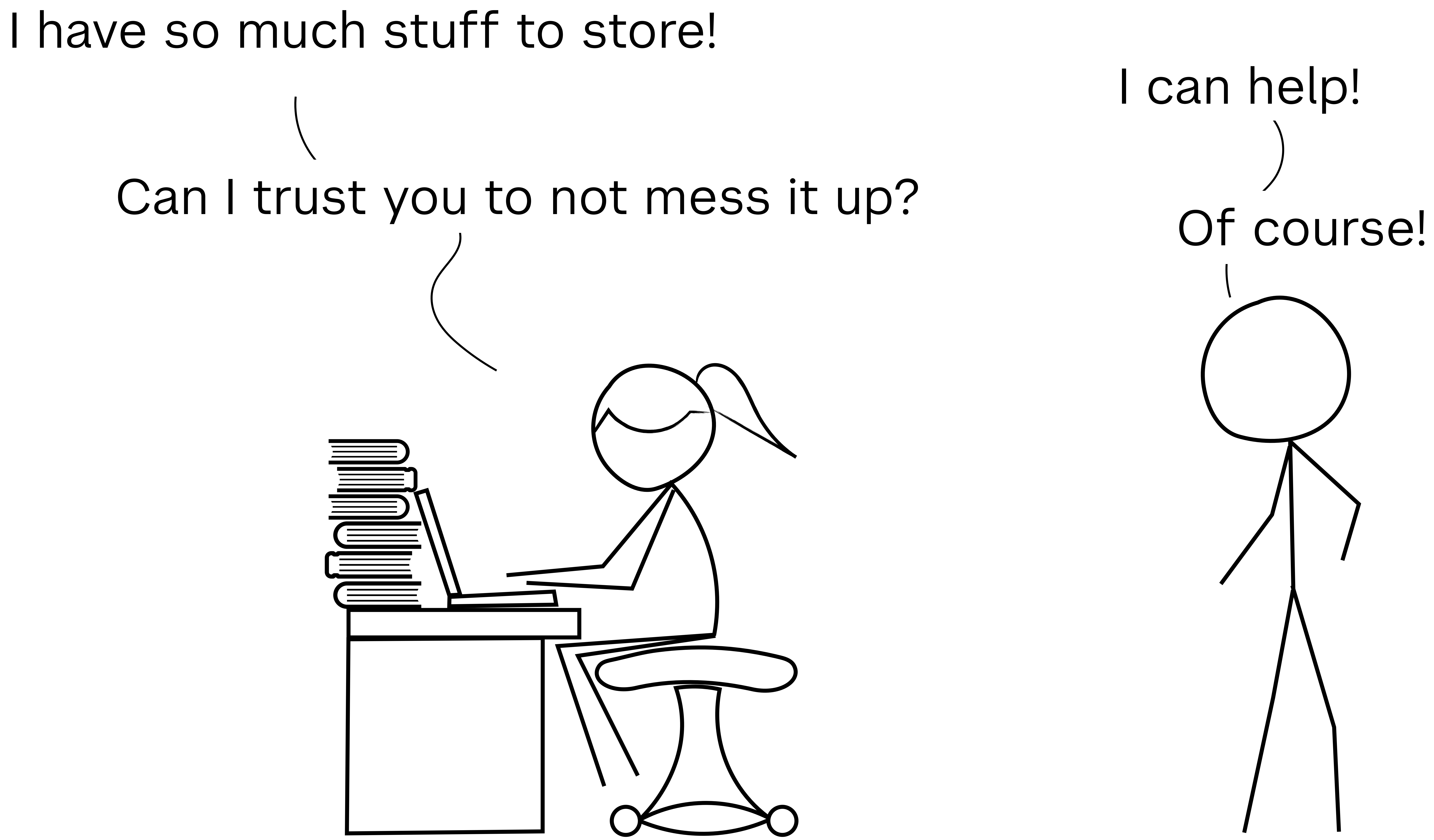
(to appear at CCS'25)

I have so much stuff to store!


I have so much stuff to store!


I have so much stuff to store!
I can help!


I have so much stuff to store!
Can I trust you to not mess it up?
I can help!


I have so much stuff to store!
Can I trust you to not mess it up?
I can help!
Of course!

How can Alice safely outsource data storage to Bob?

How can Alice safely outsource data storage to Bob?

If Alice can state her work as operations on an **authenticated data structure** then it can be outsourced to Bob, but later verified by Alice!

This is done by having Bob produce a **compact proof** that Alice can check.

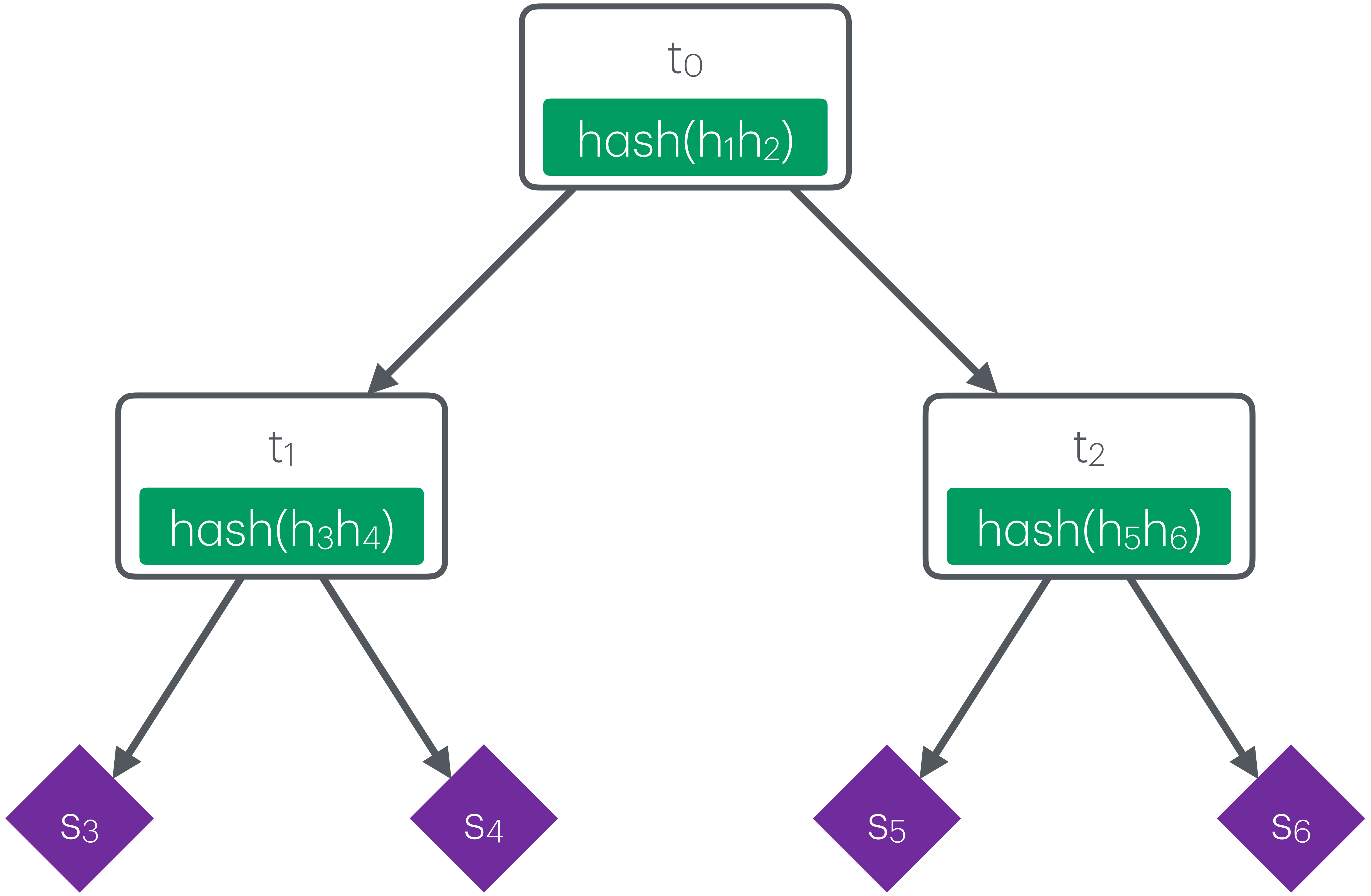
How can Alice safely outsource data storage to Bob?

If Alice can state her work as operations on an **authenticated data structure** then it can be outsourced to Bob, but later verified by Alice!

This is done by having Bob produce a **compact proof** that Alice can check.

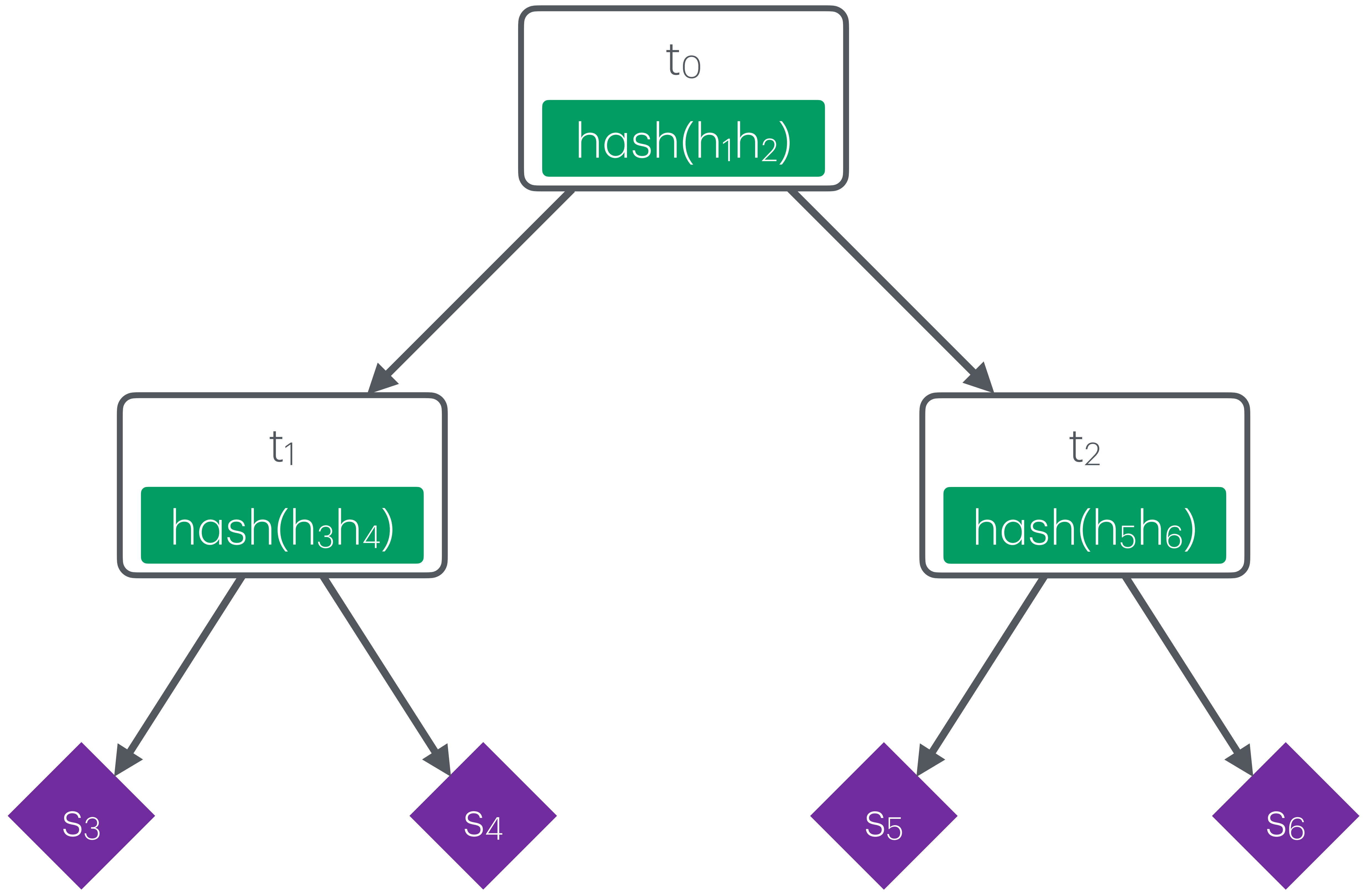
> **ADSs allow outsourcing data storage and processing tasks to untrusted parties without loss of integrity.**

Example: Merkle Tree

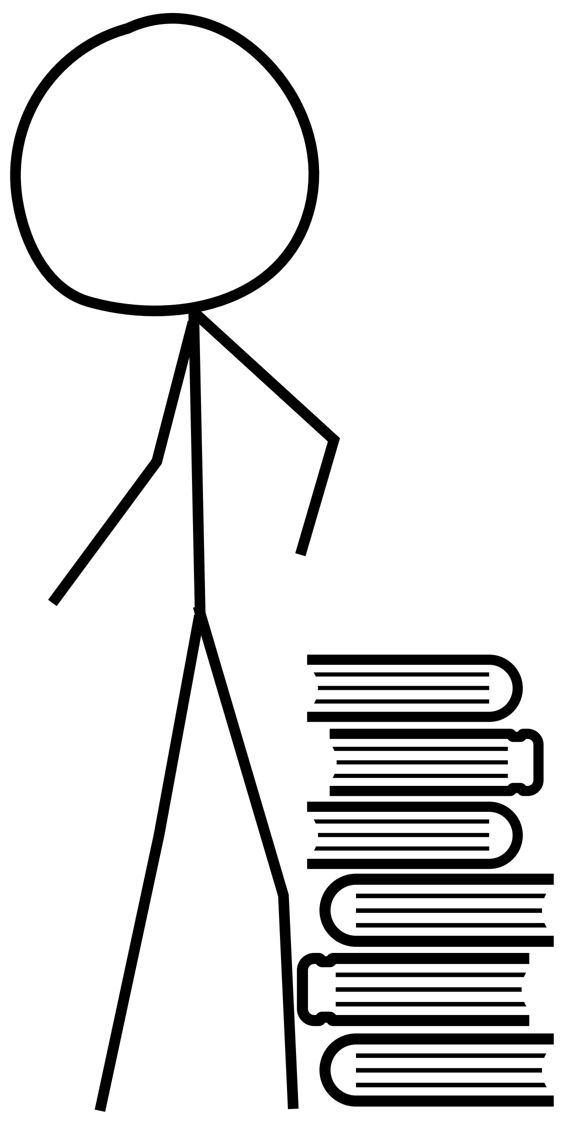


(h_i denotes the hash of t_i / s_i)

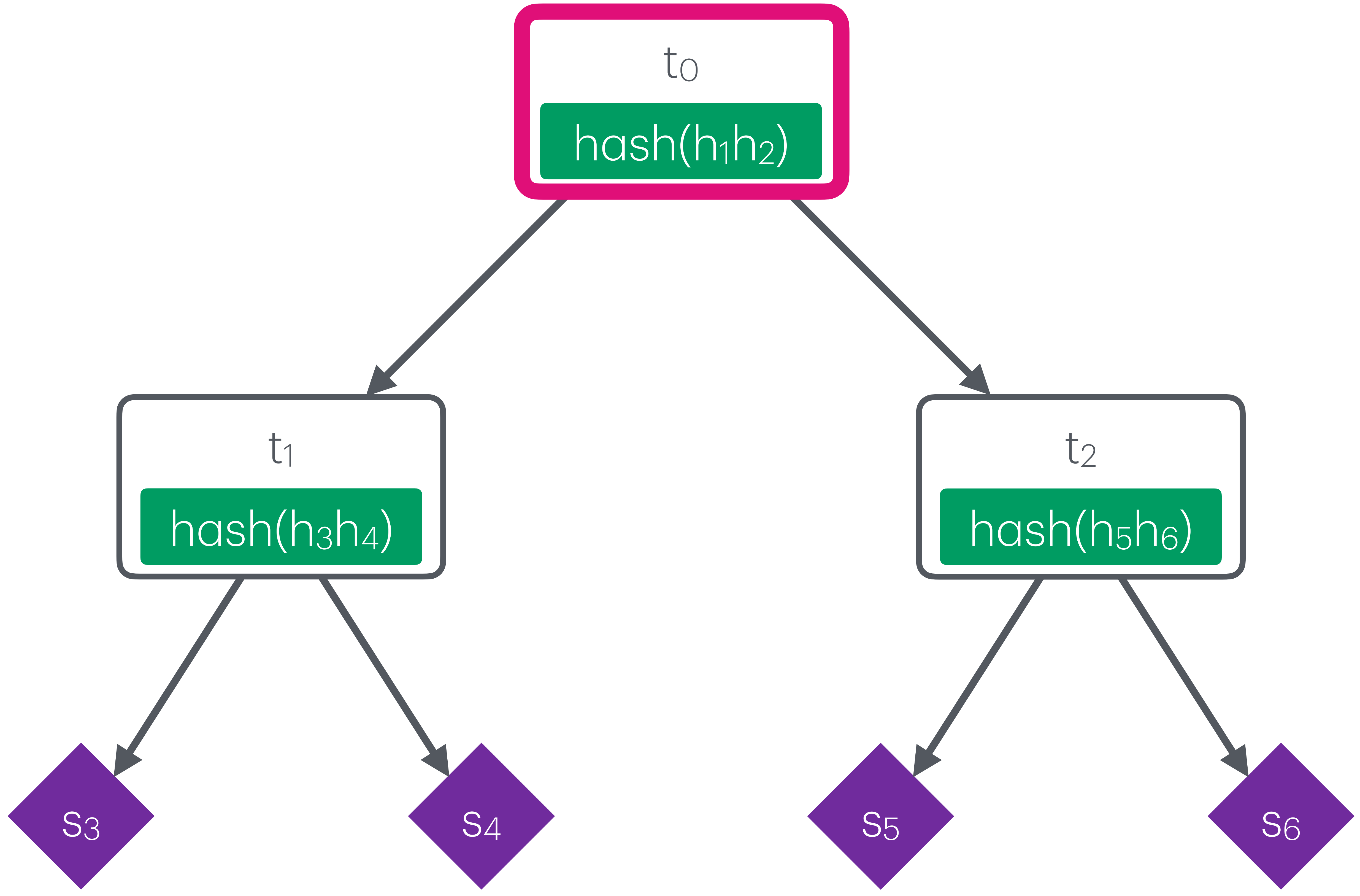
Example: Merkle Tree (Prover)



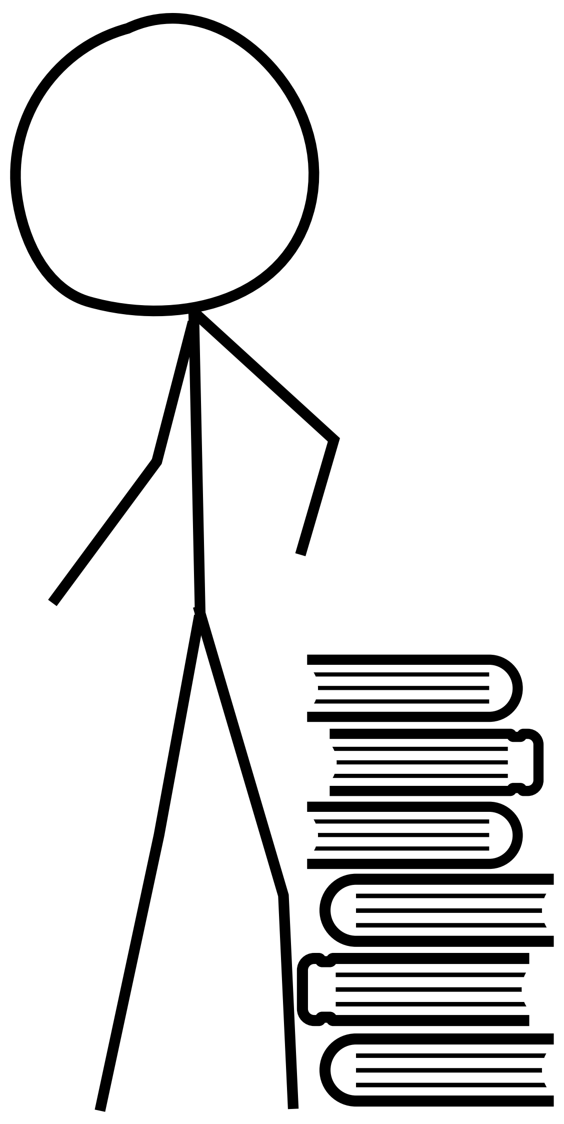
fetch([R, L], t_0) =



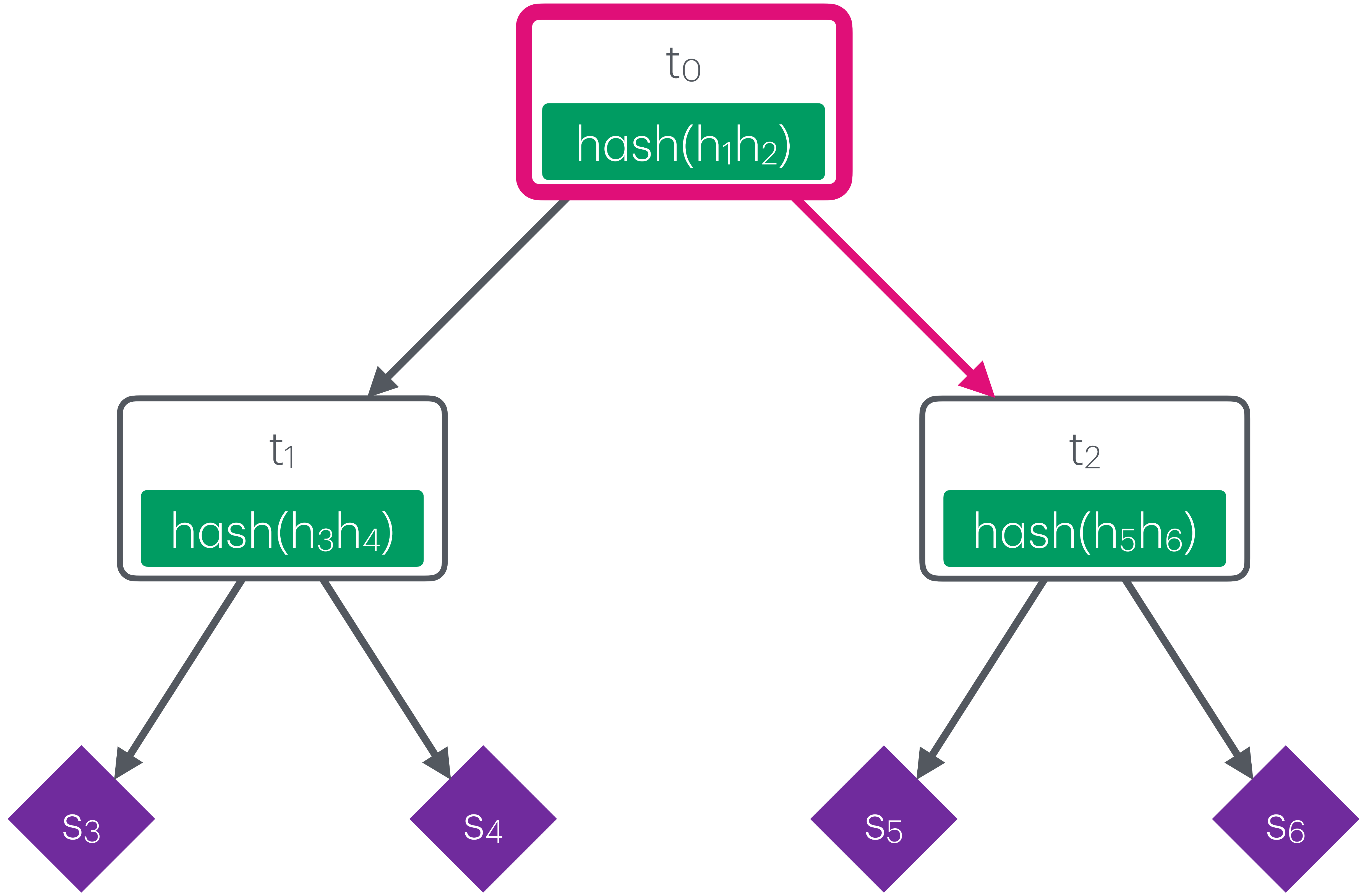
Example: Merkle Tree (Prover)



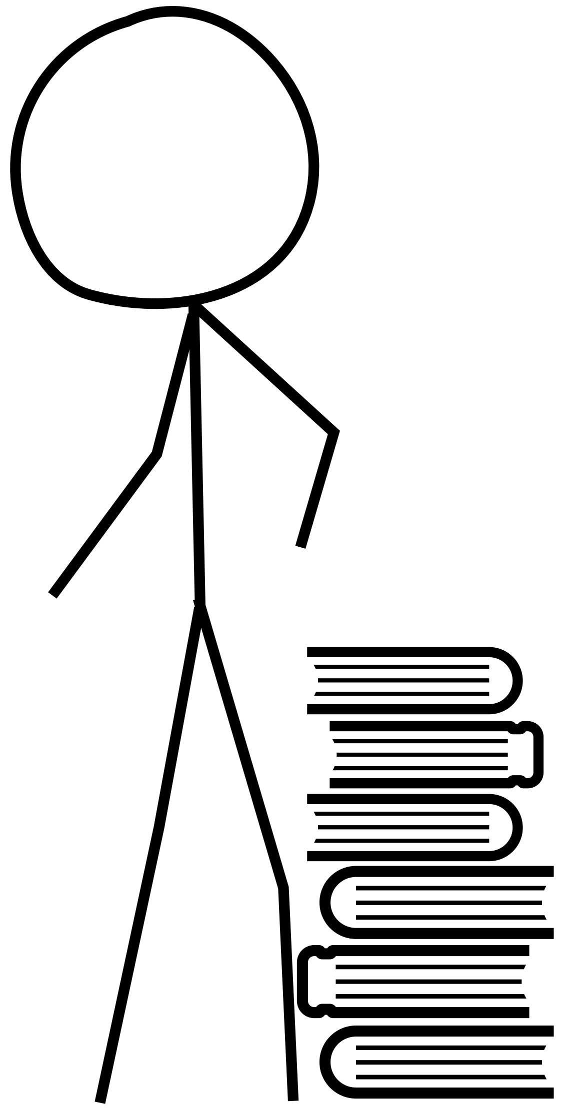
$fetch([R, L], t_0) =$



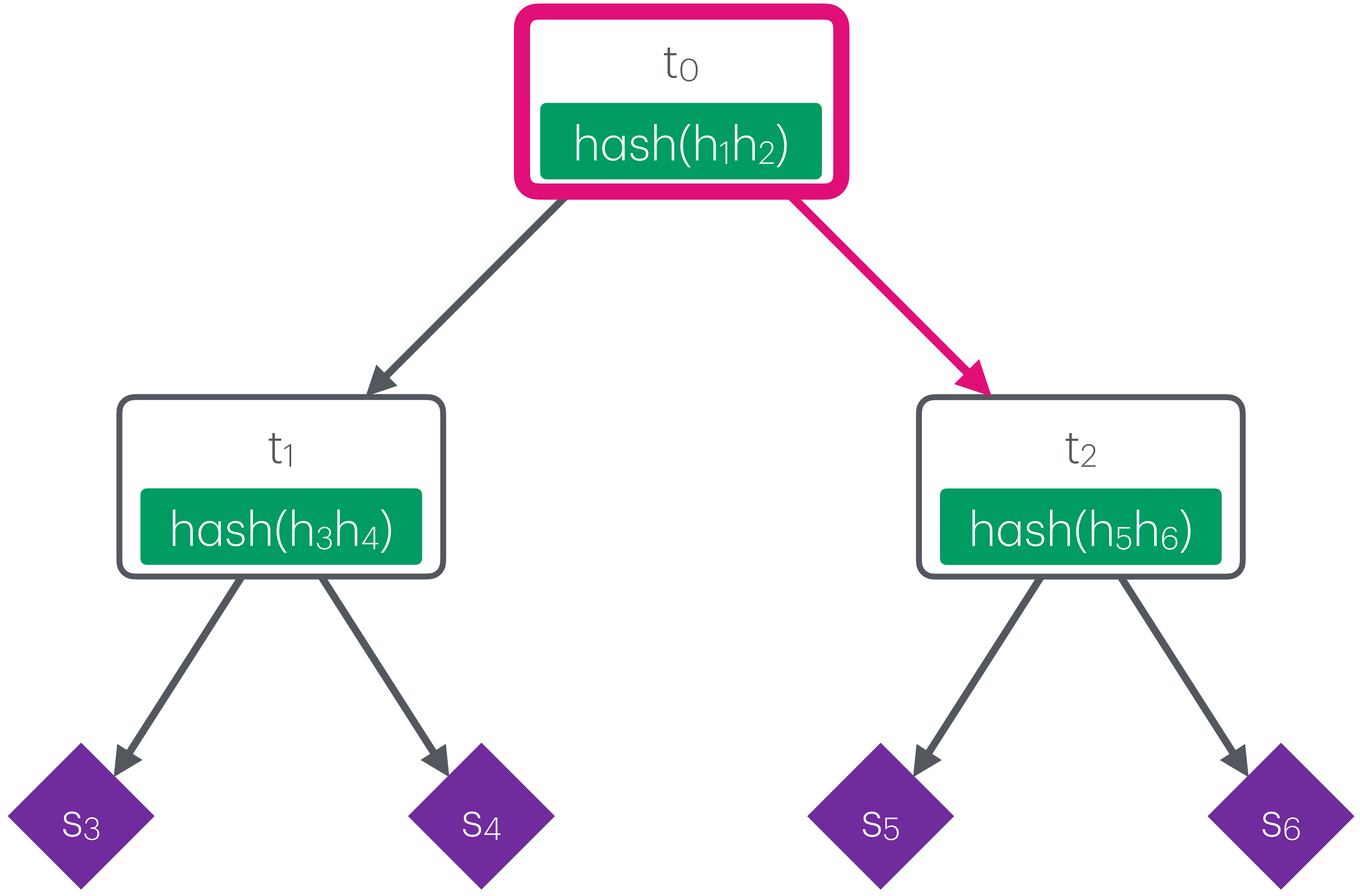
Example: Merkle Tree (Prover)



fetch([R, L], t_0) =

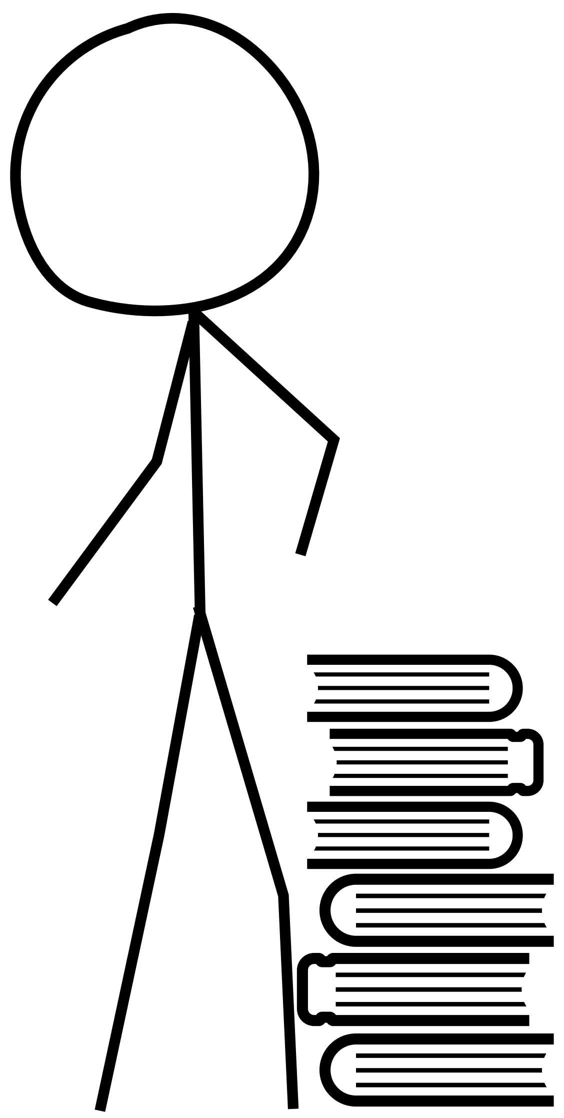


Example: Merkle Tree (Prover)

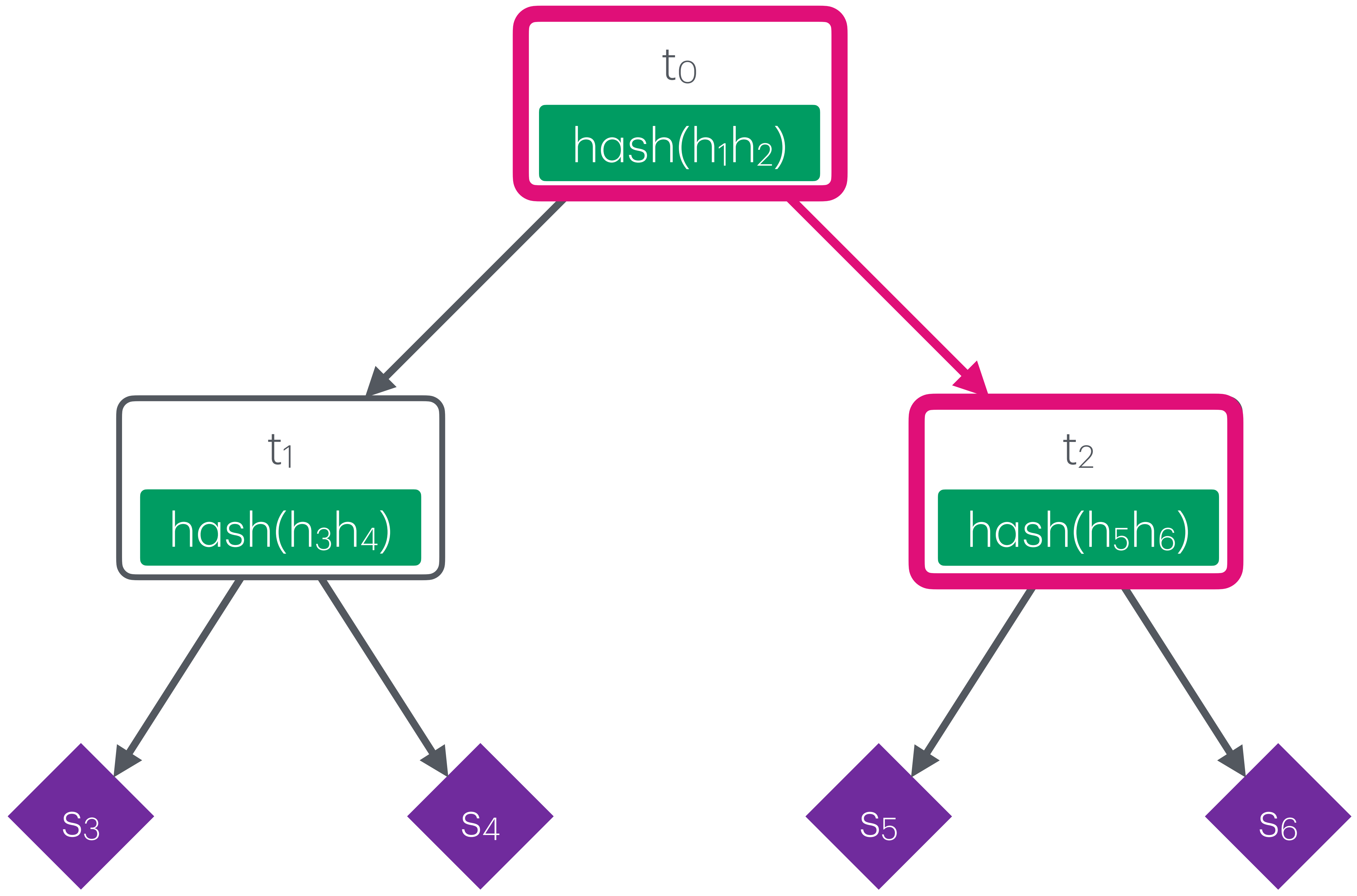


fetch([R, L], t_0) =

([h_1

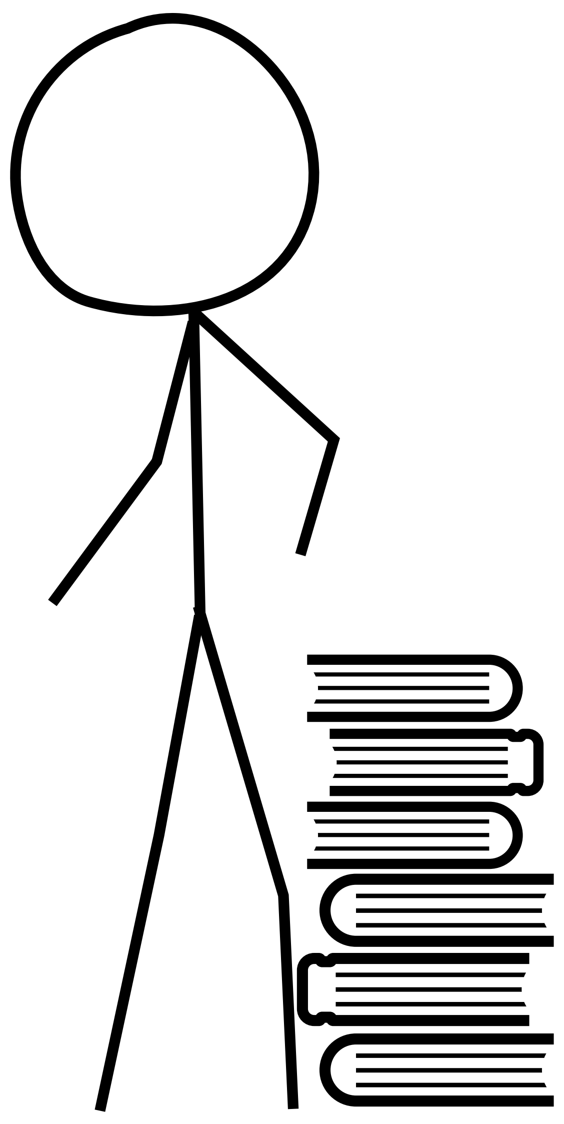


Example: Merkle Tree (Prover)

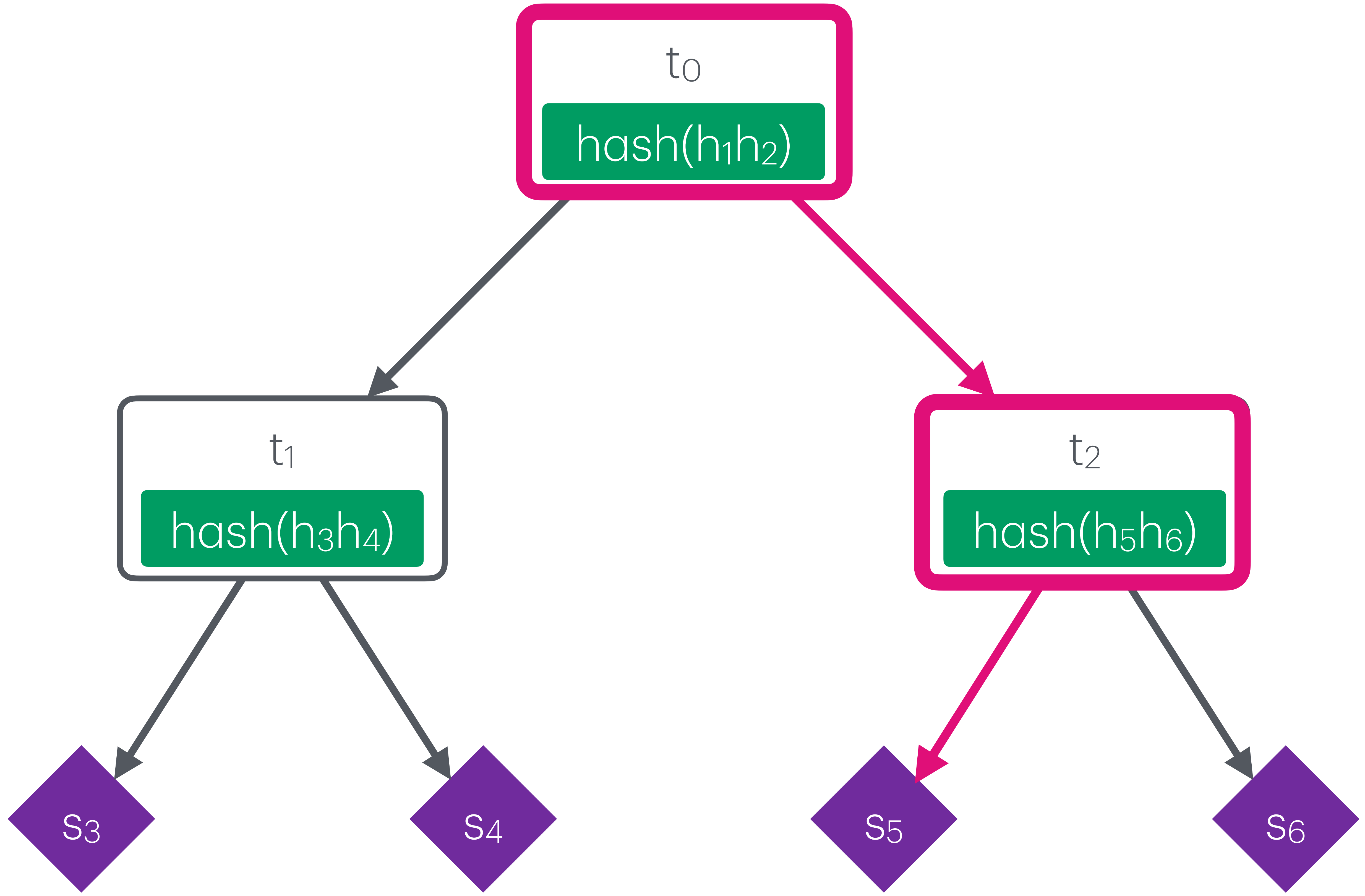


fetch([R, L], t_0) =

([h_1

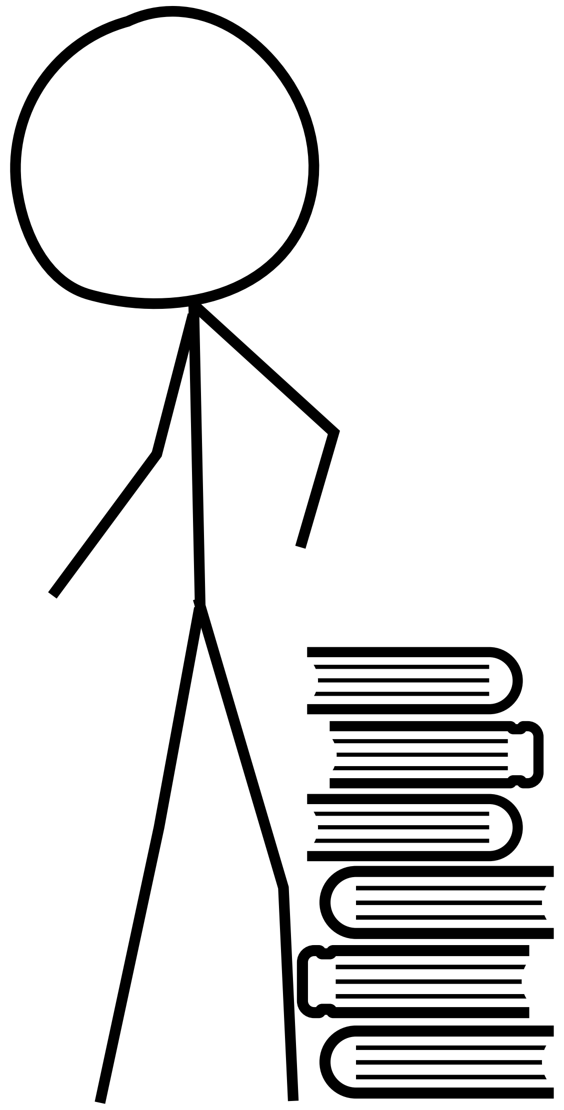


Example: Merkle Tree (Prover)

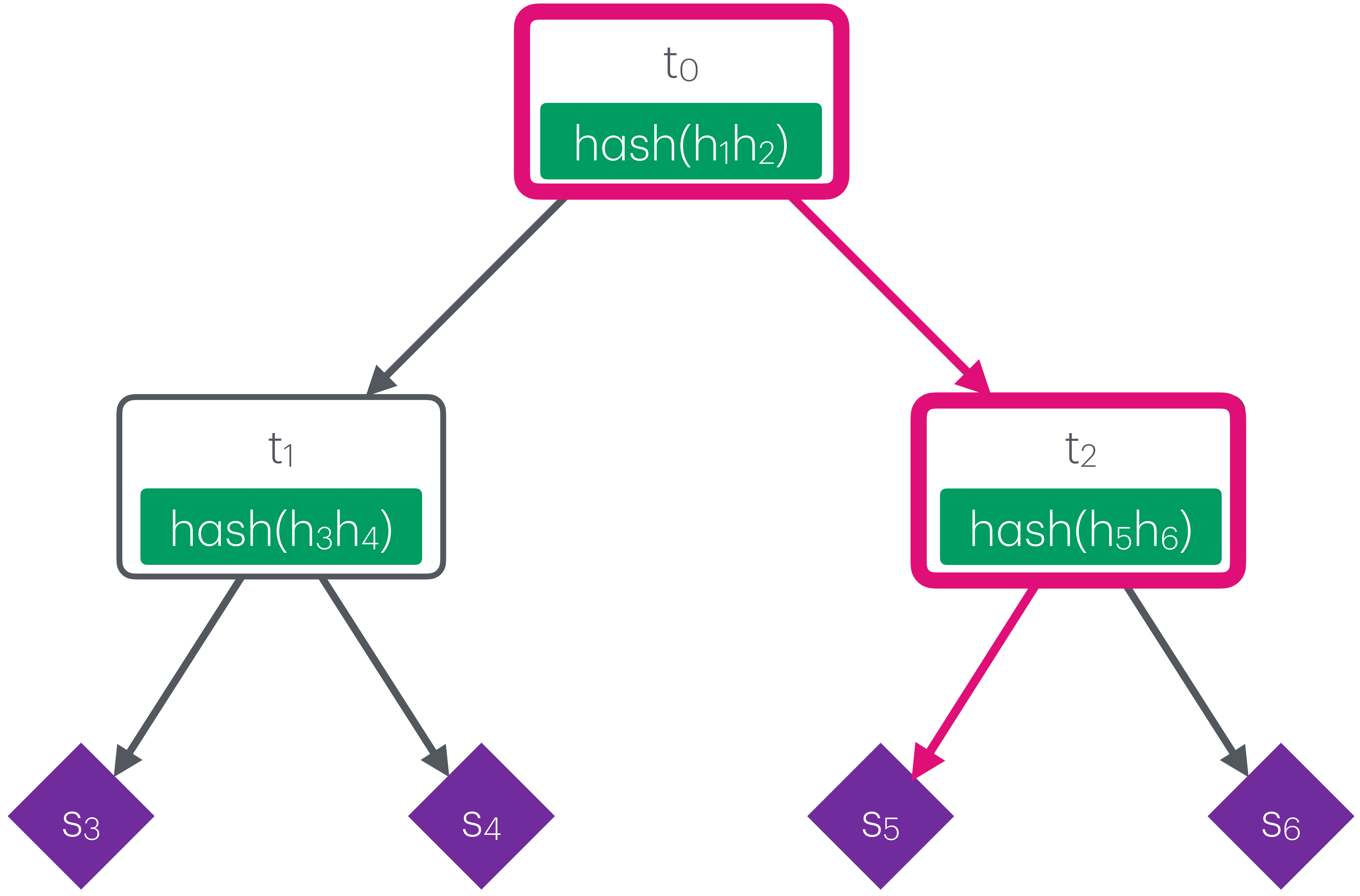


fetch([R, L], t_0) =

([h_1

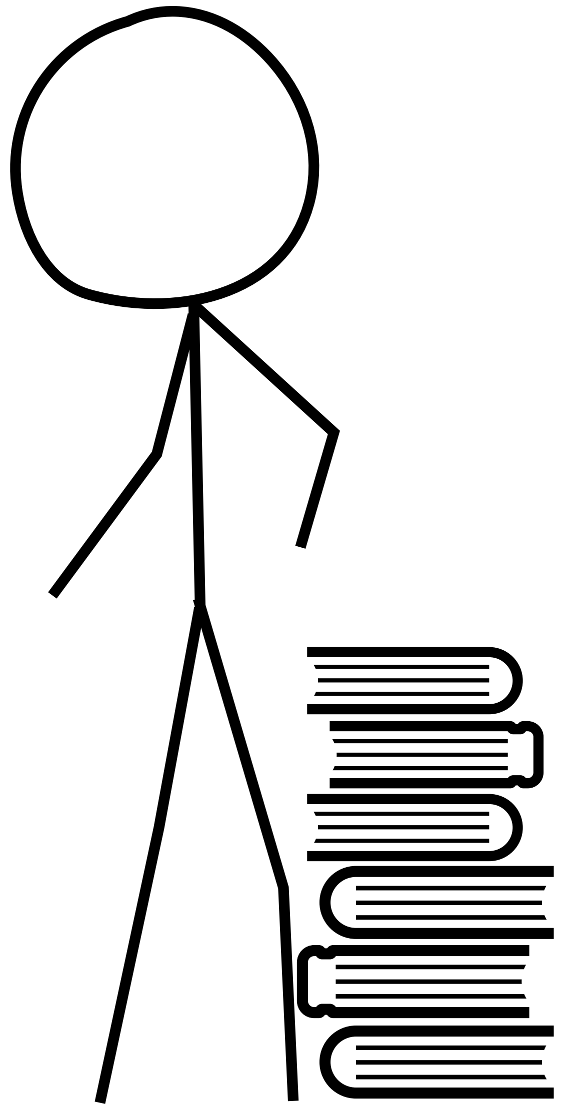


Example: Merkle Tree (Prover)

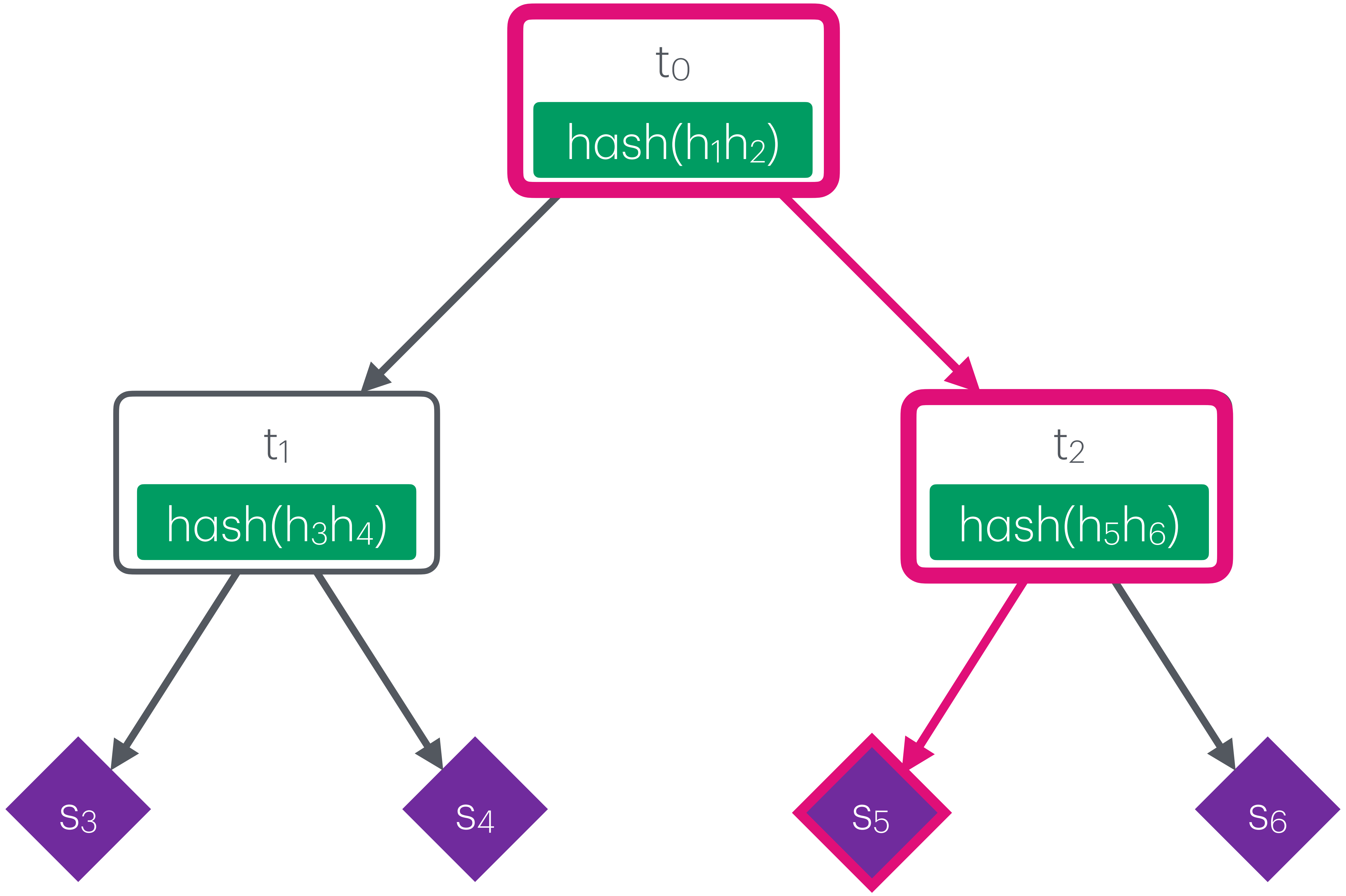


$fetch([R, L], t_0) =$

$([h_1, h_6$

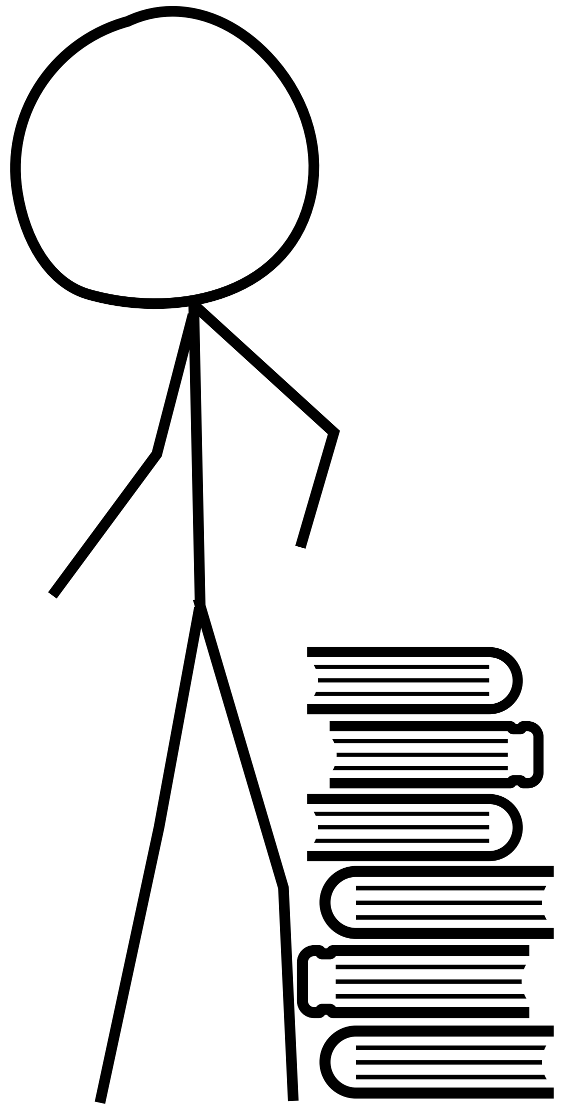


Example: Merkle Tree (Prover)

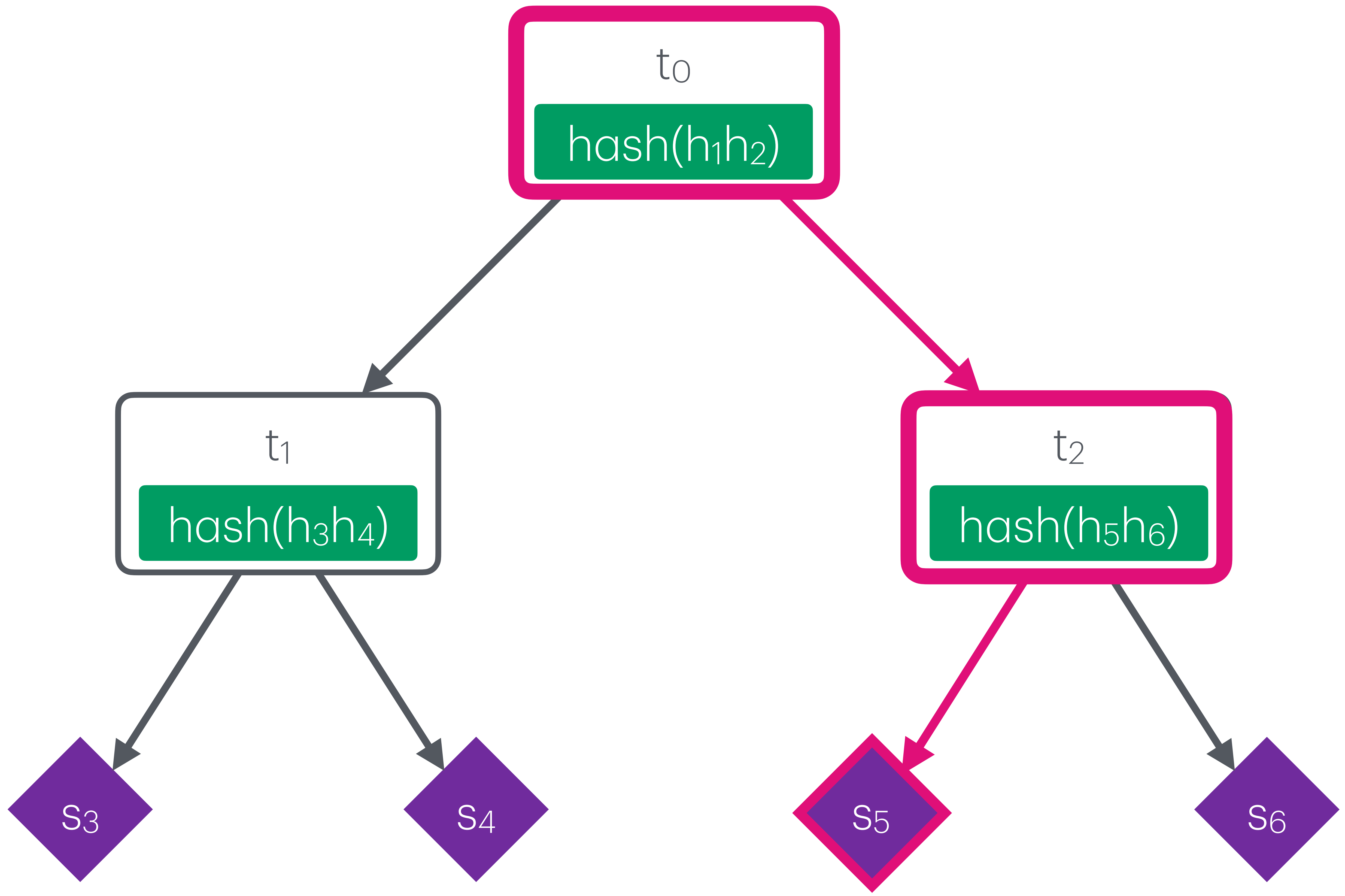


$\text{fetch}([R, L], t_0) =$

$([h_1, h_6$

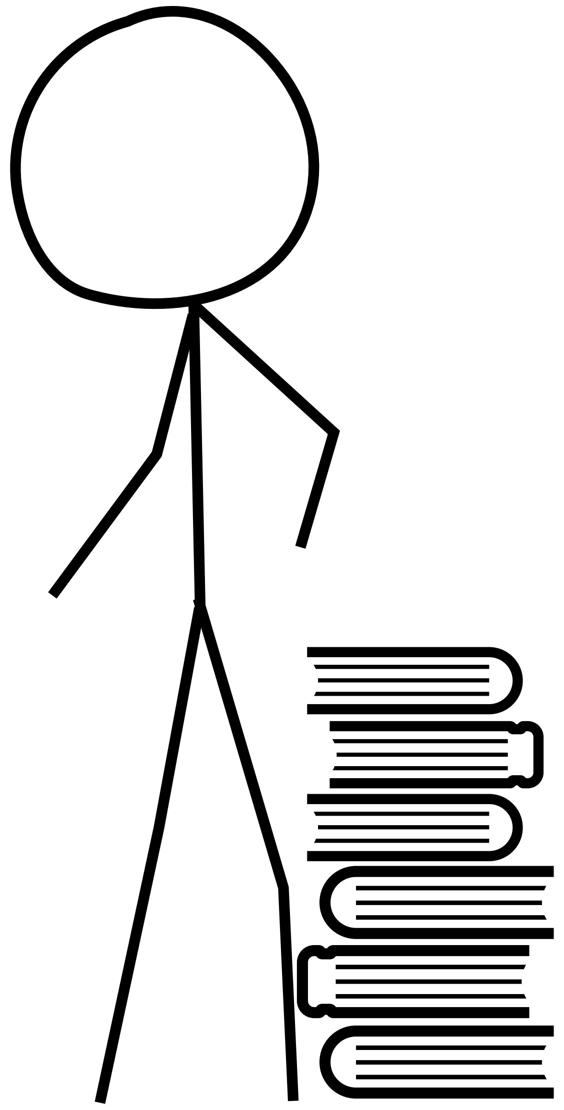


Example: Merkle Tree (Prover)

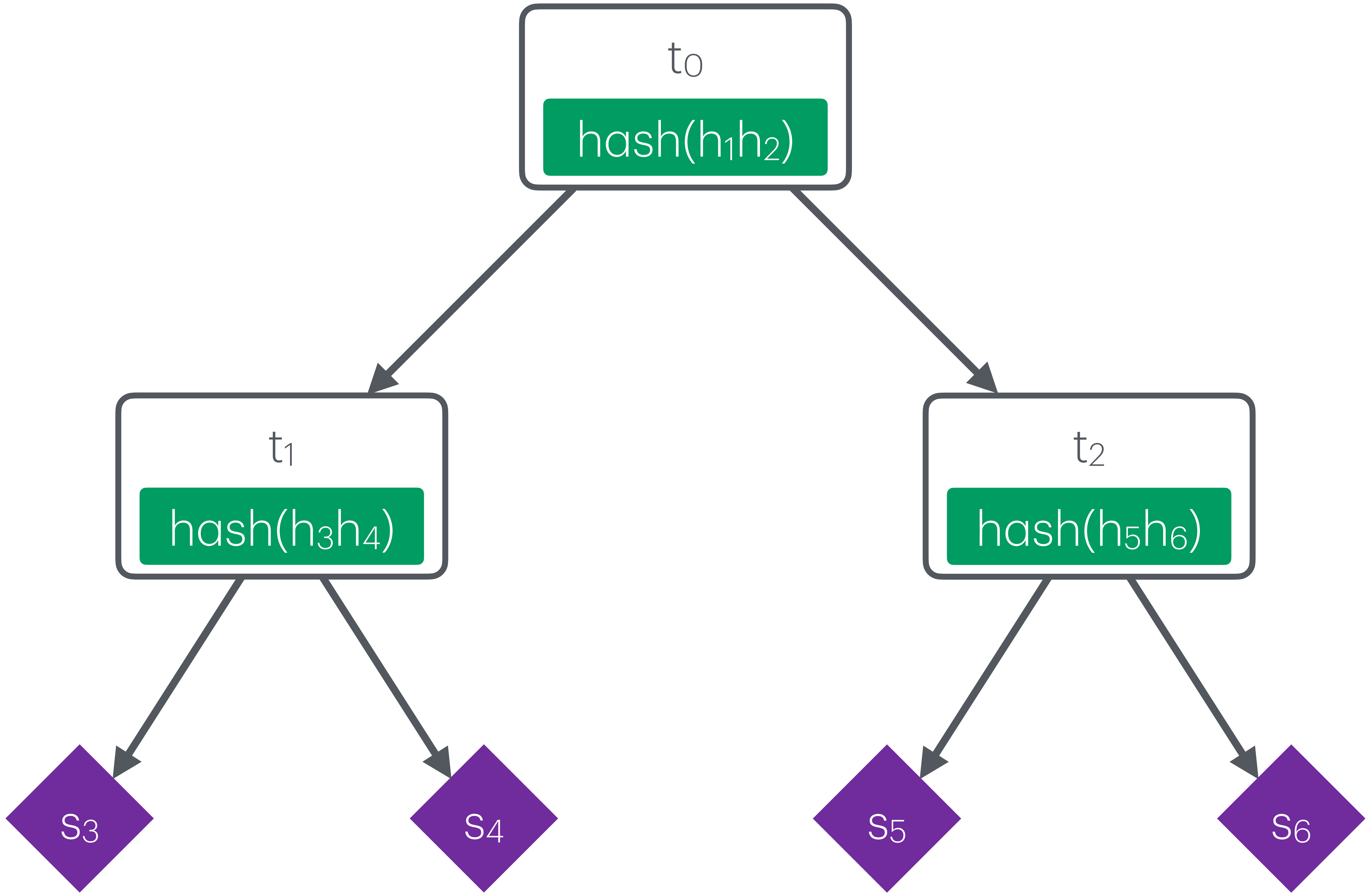


fetch([R, L], t_0) =

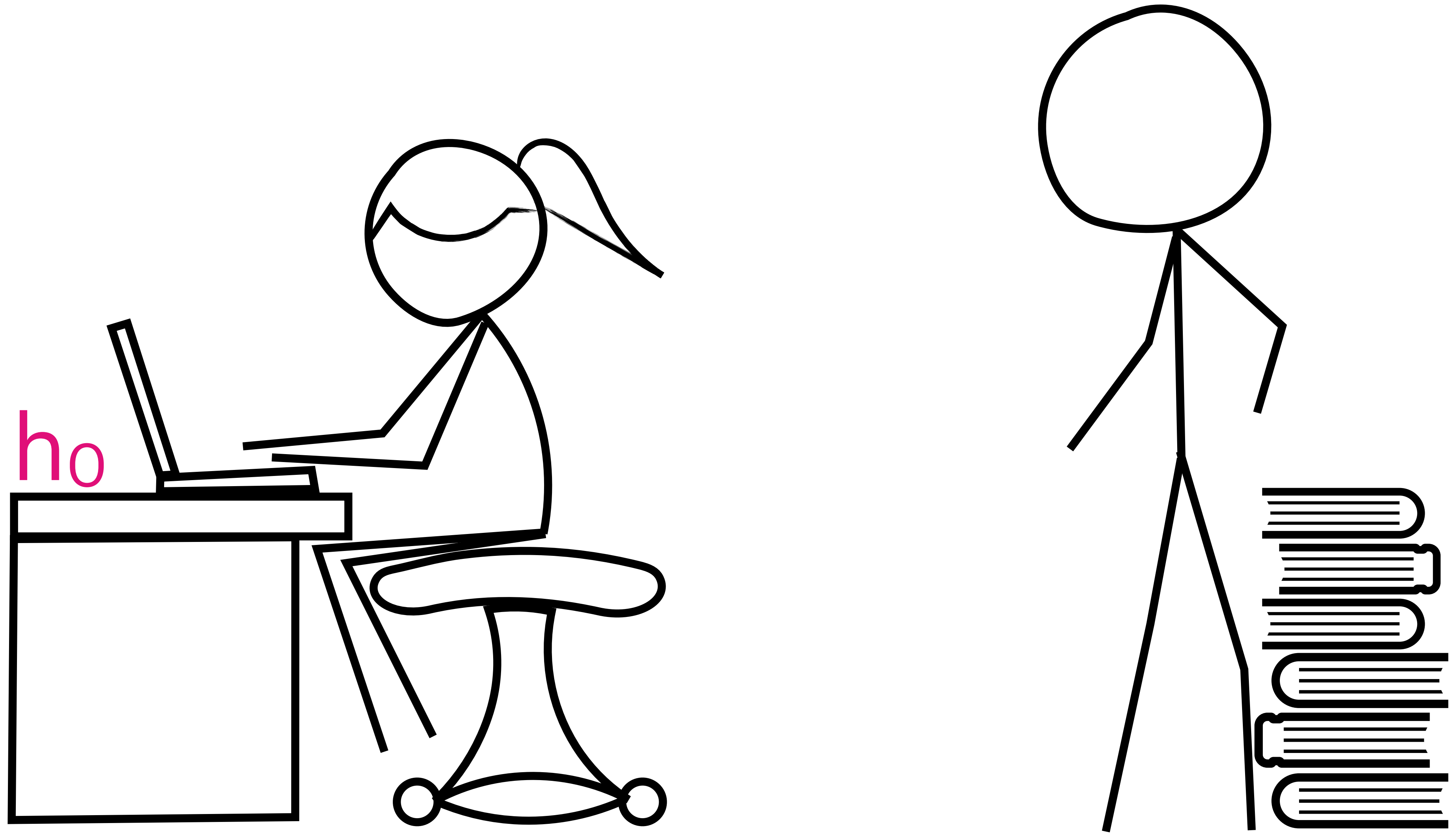
([h_1, h_6, s_5], s_5)



Example: Merkle Tree (Verifier)



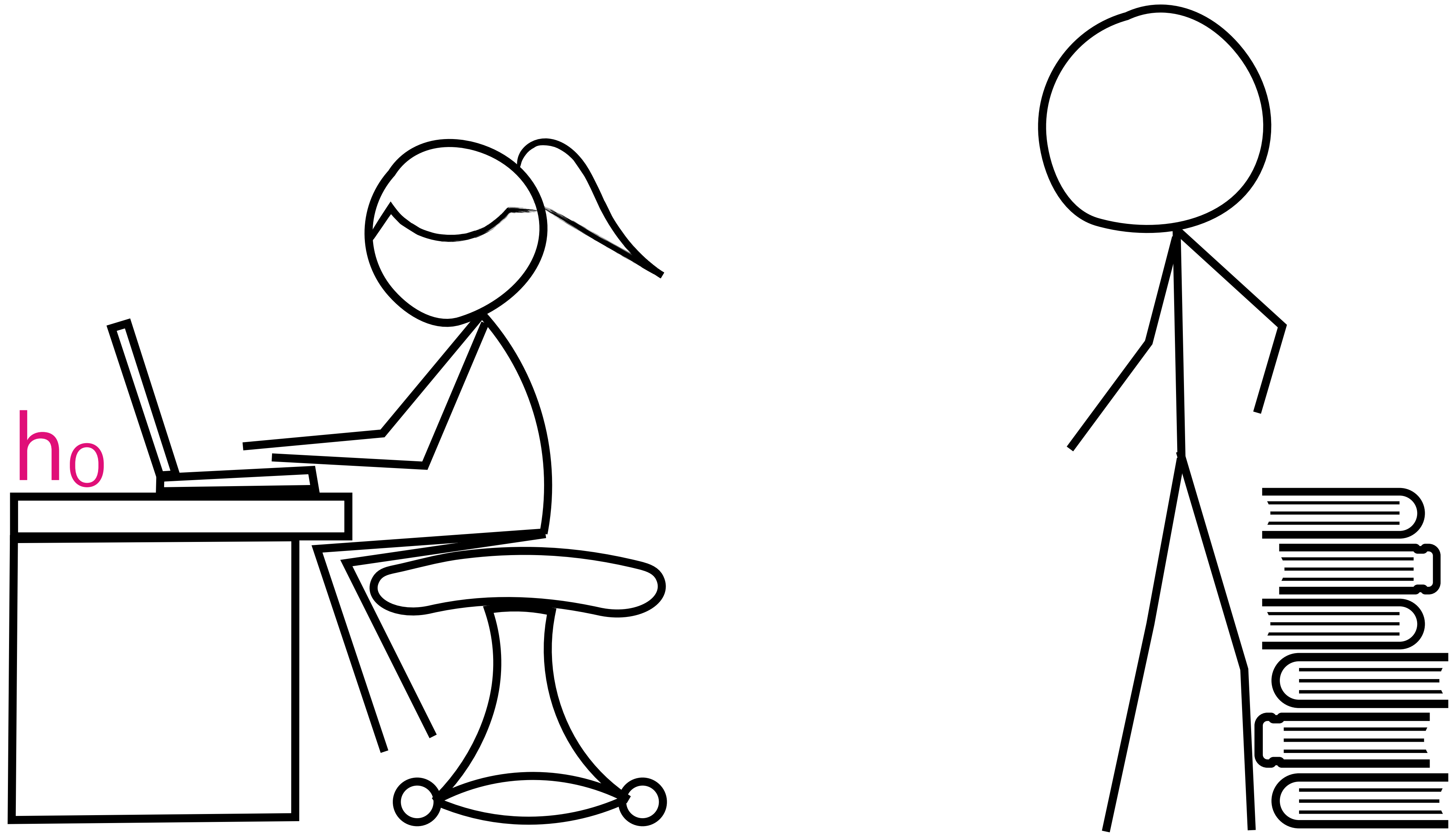
$$\text{fetch}([R, L], t_0) = ([h_1, h_6, s_5], s_5)$$



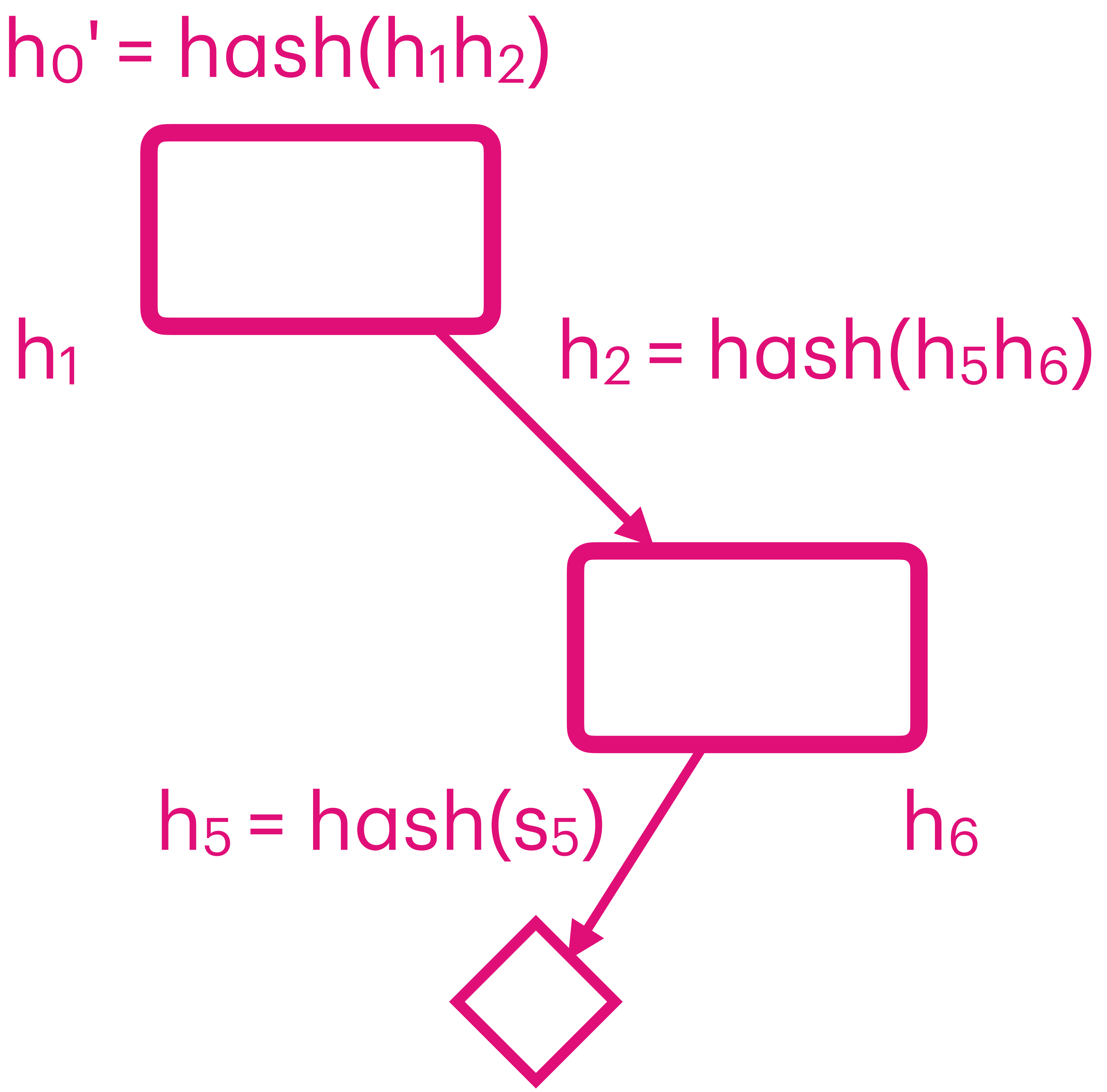
Example: Merkle Tree (Verifier)

$\text{fetch}([R, L], t_0) =$

$([h_1, h_6, s_5], s_5)$

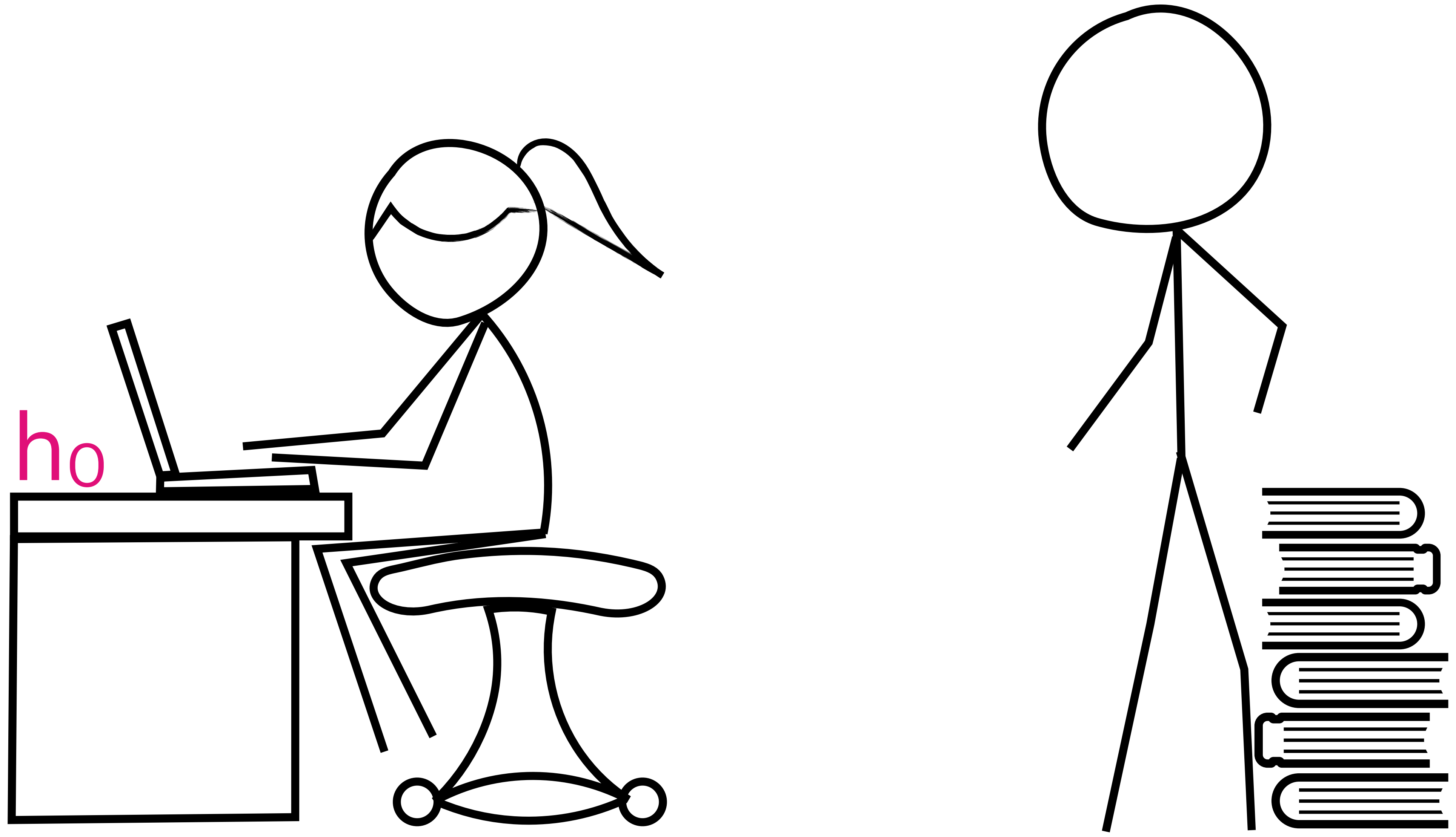


Example: Merkle Tree (Verifier)

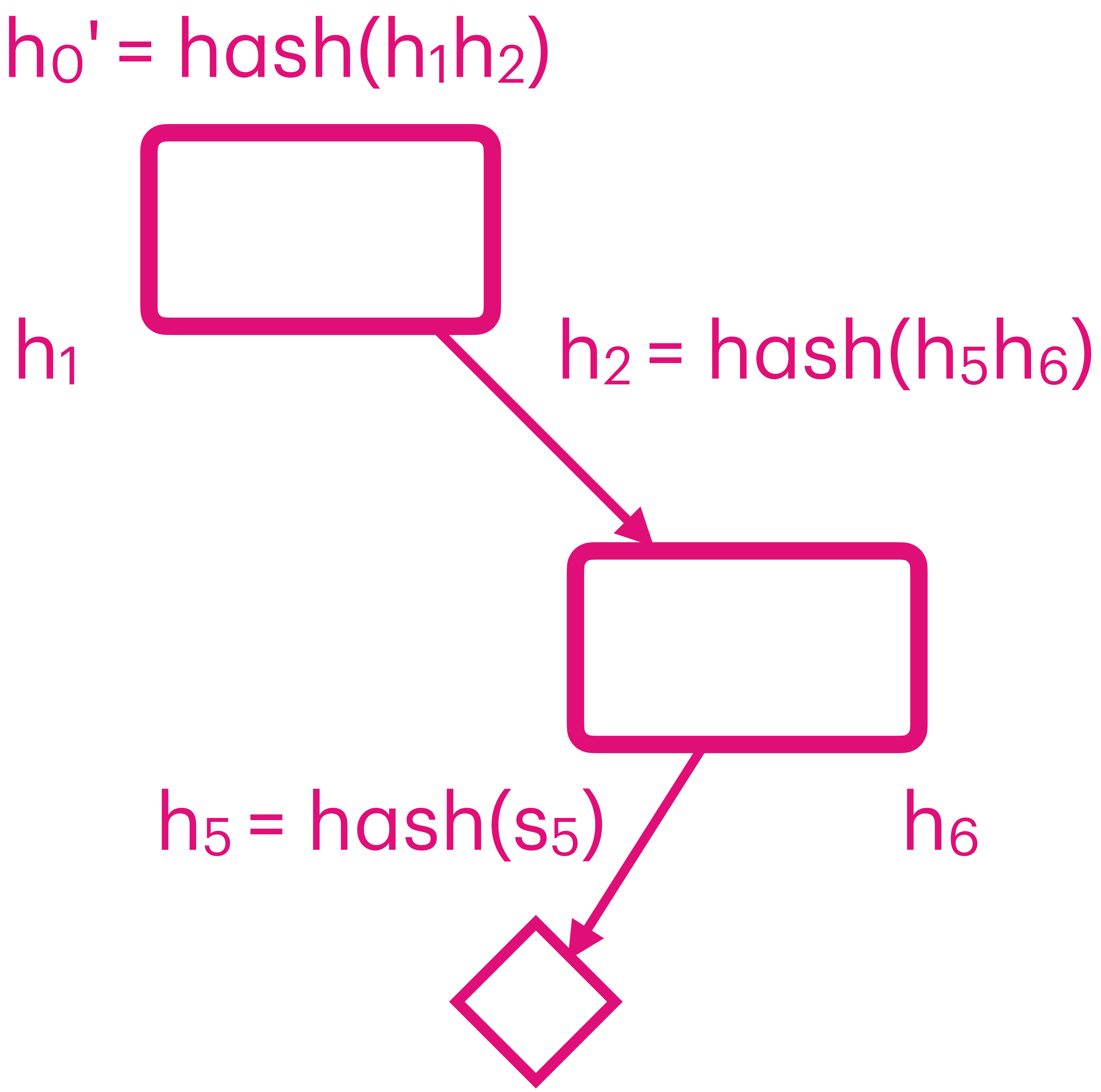


$fetch([R, L], t_0) =$

$([h_1, h_6, s_5], s_5)$

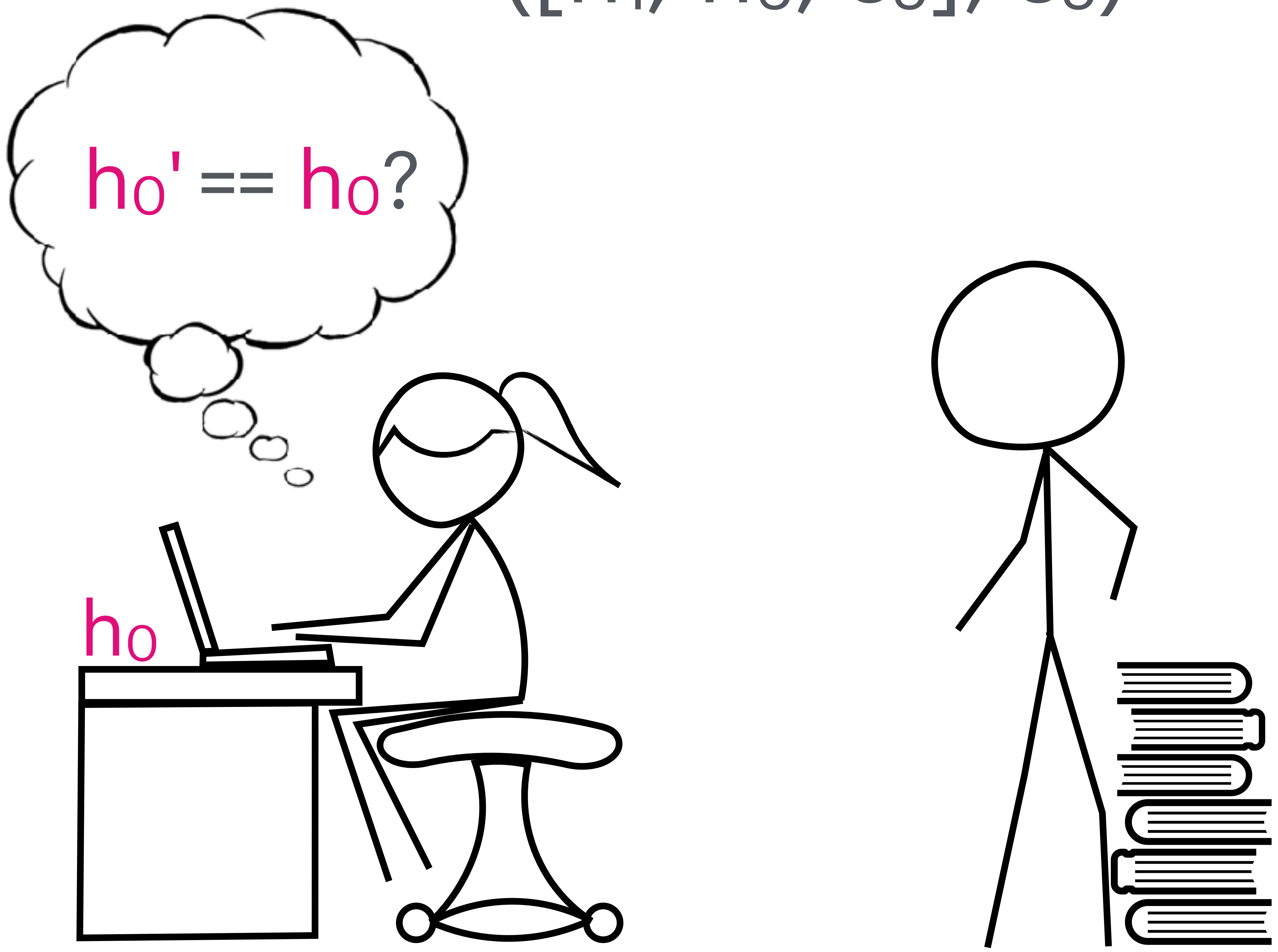


Example: Merkle Tree (Verifier)



$\text{fetch}([R, L], t_0) =$

$([h_1, h_6, s_5], s_5)$



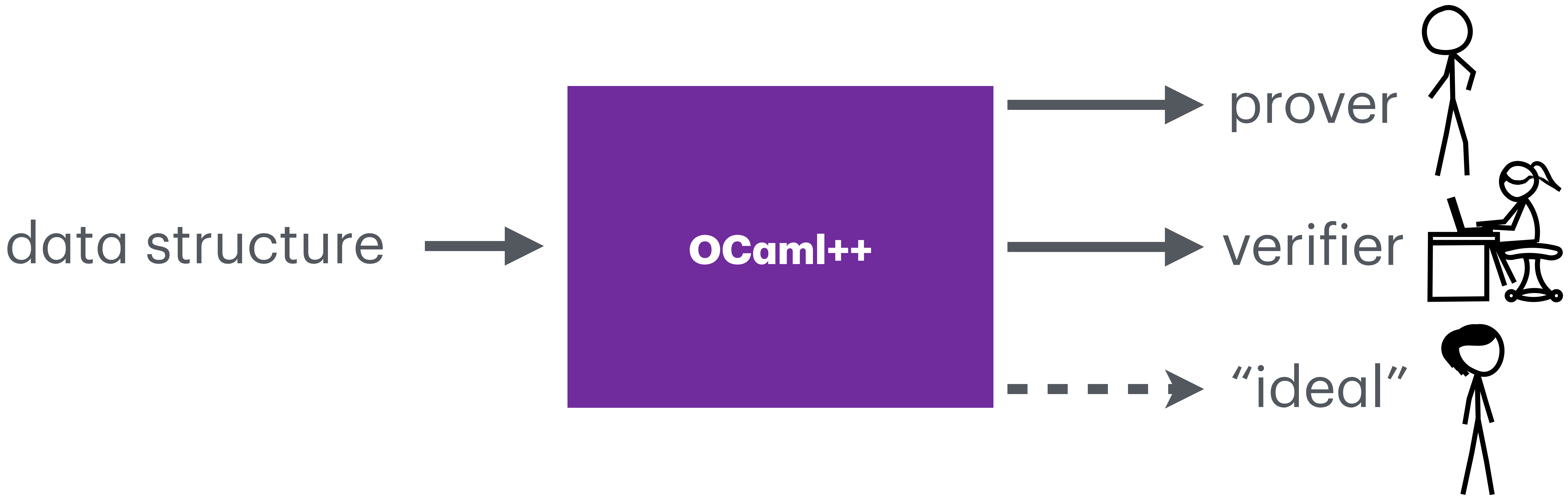
Use cases

- **Certificate transparency:** Google Chrome, Cloudflare, Let’s Encrypt, Firefox, …
- **Key transparency:** WhatsApp, Signal, …
- **Binary transparency:** Pixel Binaries, Go modules, …
- **Protection against memory corruption**
- **…**

Authenticated Data Structures, Generically

Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi
University of Maryland, College Park, USA

Miller et al. realized that the prover and verifier can be **compiled** from a single implementation of the “non-authenticated” data structure.



Miller et al.'s approach

OCaml is extended with three new primitives:

- authenticated types $\bullet\,\tau$
- auth : 'a $\rightarrow$ $\bullet$ 'a
- unauth : $\bullet$ 'a $\rightarrow$ 'a

Miller et al.'s approach

OCaml is extended with three new primitives:

- authenticated types $\bullet\,\tau$
- auth : 'a $\rightarrow \bullet$ 'a
- unauth : $\bullet$ 'a $\rightarrow$ 'a

```
type tree = Tip of string | Bin of •tree × •tree
type bit = L | R
let rec fetch (idx:bit list) (t:•tree) : string =
  match idx, unauth t with
  | [], Tip a → a
  | L :: idx, Bin(l,_) → fetch idx l
  | R :: idx, Bin(_,r) → fetch idx r
```

To justify the correctness of their approach, they define a core calculus and show **security** and **correctness**:

To justify the correctness of their approach, they define a core calculus and show **security** and **correctness**:

Security: If the **verifier** accepts a proof p and returns v then

- the **ideal** execution returns v or
- a hash collision occurred.

To justify the correctness of their approach, they define a core calculus and show **security** and **correctness**:

Security: If the **verifier** accepts a proof p and returns v then

- the **ideal** execution returns v or
- a hash collision occurred.

Correctness: If the **prover** generates a proof p and a result v then

- the **ideal** execution returns v and
- the **verifier** accepts p and returns v as well.

Limitations

Limitations

1. A custom compiler frontend imposes development burden.

Limitations

1. A custom compiler frontend imposes development burden.

2. The compiler implements several optimizations that are not covered by the security and correctness theorems.

Limitations

1. A custom compiler frontend imposes development burden.
2. The compiler implements several optimizations that are not covered by the security and correctness theorems.
3. The generated data structures are not always as efficient or produce proofs as compact as hand-written implementations.

About Blog Publications

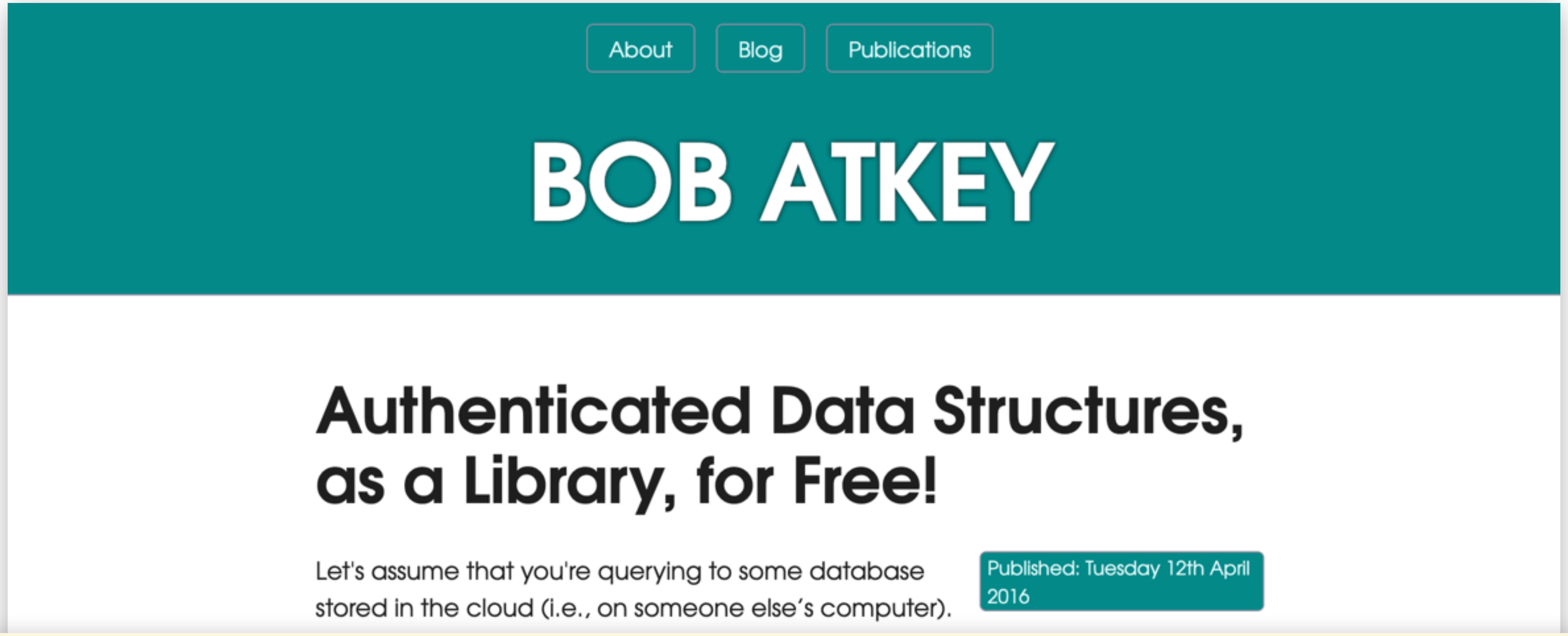
BOB ATKEY

Authenticated Data Structures, as a Library, for Free!

Published: Tuesday 12th April 2016

Let's assume that you're querying to some database stored in the cloud (i.e., on someone else's computer). Being of a sceptical mind, you worry whether or not the answers you get back are from the database you expect. Or is the cloud lying to you?

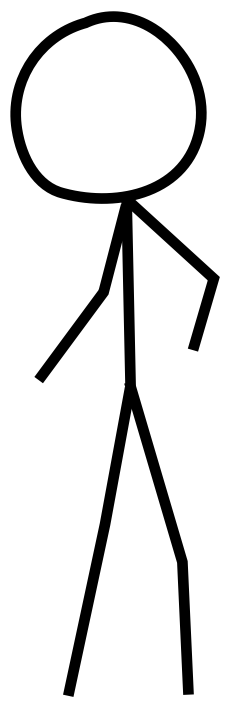
Authenticated Data Structures (ADSs) are a proposed solution to this problem. When the server sends back its answers, it also sends back a "proof" that the answer came from the database it claims. You, the client, verify this proof. If the proof doesn't verify, then you've got evidence that the server was lying. If the




```
module Merkle = functor (A : AUTHENTIKIT) -> struct
  open A

  (* ... *)

  val fetch : path -> tree auth -> string option auth_computation = (* ... *)
end
```

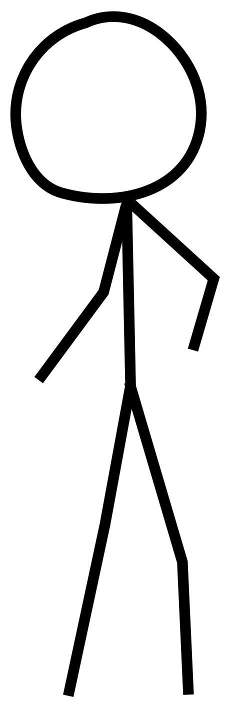
```
module Merkle = functor (A : AUTHENTIKIT) -> struct
  open A

  (* ... *)

  val fetch : path -> tree auth -> string option auth_computation = (* ... *)
end
```

```
module Prover : AUTHENTIKIT
```



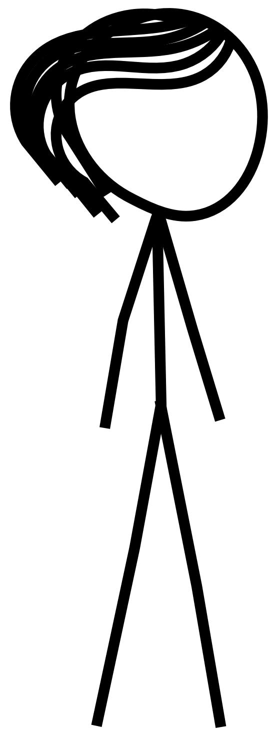
```
module Merkle = functor (A : AUTHENTIKIT) -> struct
  open A

  (* ... *)

  val fetch : path -> tree auth -> string option auth_computation = (* ... *)
end
```

```
module Prover : AUTHENTIKIT
```



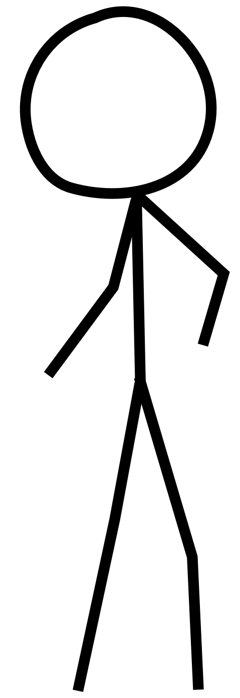
```
module Merkle = functor (A : AUTHENTIKIT) -> struct
  open A

  (* ... *)

  val fetch : path -> tree auth -> string option auth_computation = (* ... *)
end
```

```
module Verifier : AUTHENTIKIT
```



```
module Ideal : AUTHENTIKIT
```

```
module Prover : AUTHENTIKIT
```



```
module Merkle = functor (A : AUTHENTIKIT) -> struct
  open A

  (* ... *)

  val fetch : path -> tree auth -> string option auth_computation = (* ... *)
end
```

```
module Verifier : AUTHENTIKIT
```

This work

- Two **logical relations** and a proof of security and correctness of the Authentikit module functor construction in OCaml.
- We address the remaining two limitations:
  - We verify several of the **optimizations** supported by the compiler.
  - We show how to **safely link** manually verified code with code automatically generated by Authentikit through semantic typing.
- Full mechanization in the Rocq theorem prover.

```
module type AUTHENTIKIT = sig
  type 'a auth


  (* ... *)


  module Serializable : sig
    type 'a evidence


    (* ... *)



  end

  val auth   : 'a Serializable.evidence -> 'a -> 'a auth
  val unauth : 'a Serializable.evidence -> 'a auth -> 'a auth_computation
end
```

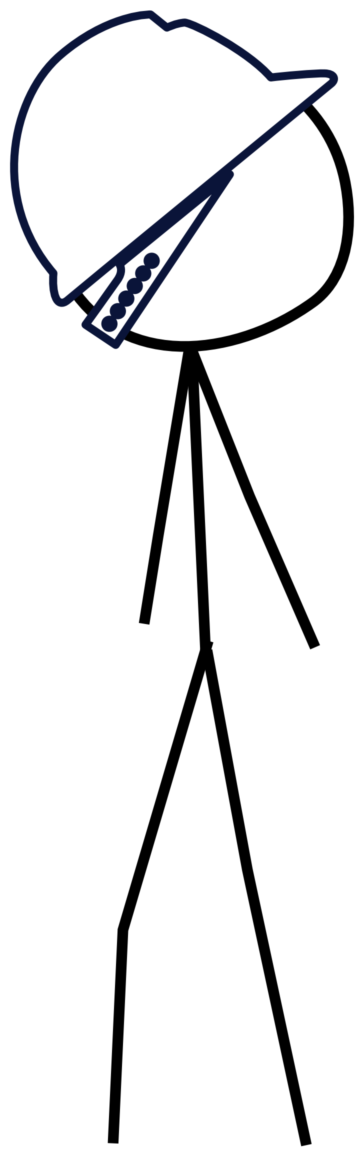
```ocaml
module type AUTHENTIKIT = sig
  type 'a auth
  type 'a auth_computation

  val return : 'a -> 'a auth_computation
  val bind   : 'a auth_computation -> ('a -> 'b auth_computation) -> 'b auth_computation

  module Serializable : sig
    type 'a evidence

    (* ... *)

  end

  val auth   : 'a Serializable.evidence -> 'a -> 'a auth
  val unauth : 'a Serializable.evidence -> 'a auth -> 'a auth_computation
end
```

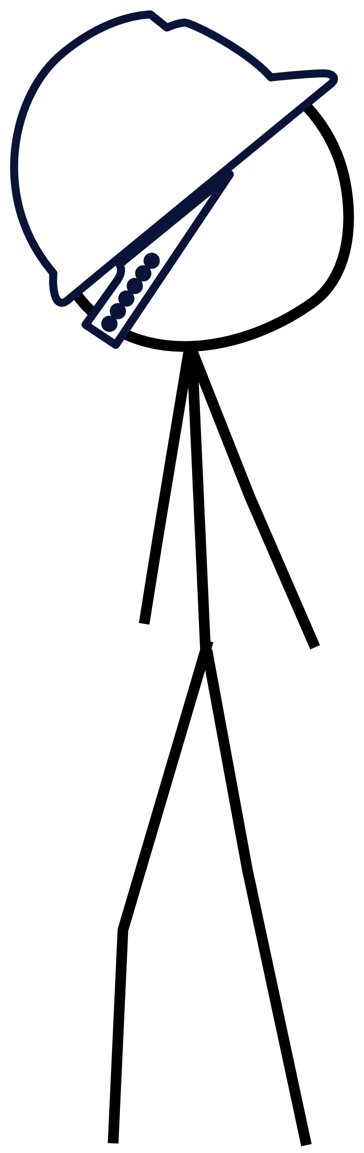
```
module Merkle = functor (A : AUTHENTIKIT) -> struct
  open A

  type path = [`L | `R] list
  type tree = [`leaf of string | `node of tree auth * tree auth]



  (* ... *)





  (* ... *)




end
```

```
module Merkle = functor (A : AUTHENTIKIT) -> struct
  open A

  type path = [`L | `R] list
  type tree = [`leaf of string | `node of tree auth * tree auth]

  let tree_evi : tree Serializable.evidence = (* ... *)

  let make_leaf (s : string) : tree auth = auth tree_evi (`leaf s)
  let make_branch (l r : tree auth) : tree auth = auth tree_evi (`node (l, r))



  (* ... *)



end
```

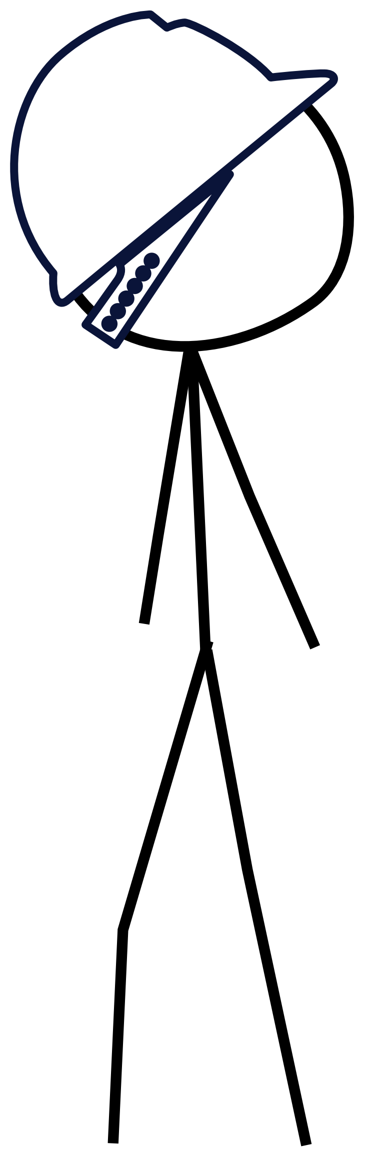
```ocaml
module Merkle = functor (A : AUTHENTIKIT) -> struct
  open A

  type path = [`L | `R] list
  type tree = [`leaf of string | `node of tree auth * tree auth]

  let tree_evi : tree Serializable.evidence = (* ... *)

  let make_leaf (s : string) : tree auth = auth tree_evi (`leaf s)
  let make_branch (l r : tree auth) : tree auth = auth tree_evi (`node (l, r))

  let rec fetch (p : path) (t : tree auth) : string option auth_computation =
    bind (unauth tree_evi t) (fun t ->
      match p, t with
      | [], `leaf s -> return (Some s)
      | `L :: p, `node (l, _) -> fetch p l
      | `R :: p, `node (_, r) -> fetch p r
      | _, _ -> return None)
end
```

```ocaml
type proof = string list

module Prover : AUTHENTIKIT =
  type 'a auth = 'a * string
  type 'a auth_computation = () -> proof * 'a







  (* ... *)







  (* ... *)

end
```



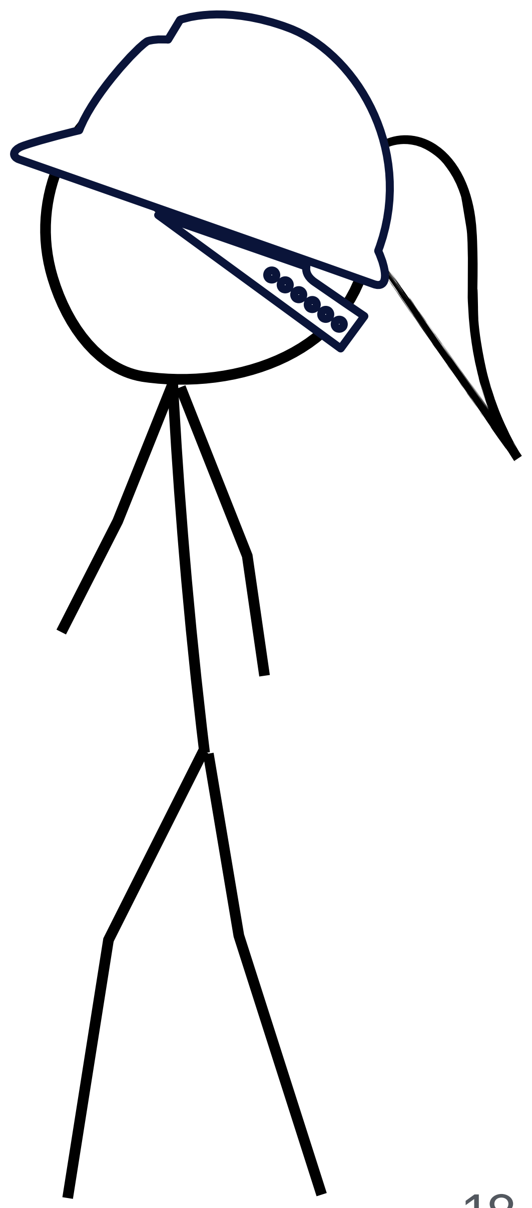


```
type proof = string list

module Prover : AUTHENTIKIT =
  type 'a auth = 'a * string
  type 'a auth_computation = () -> proof * 'a

  let return a () = ([], a)
  let bind c f =
    let (prf,  a) = c () in
    let (prf', b) = f a () in
    (prf @ prf', b)

  module Serializable = struct
    type 'a evidence = 'a -> string

    (* ... *)
  end

  (* ... *)

end
```

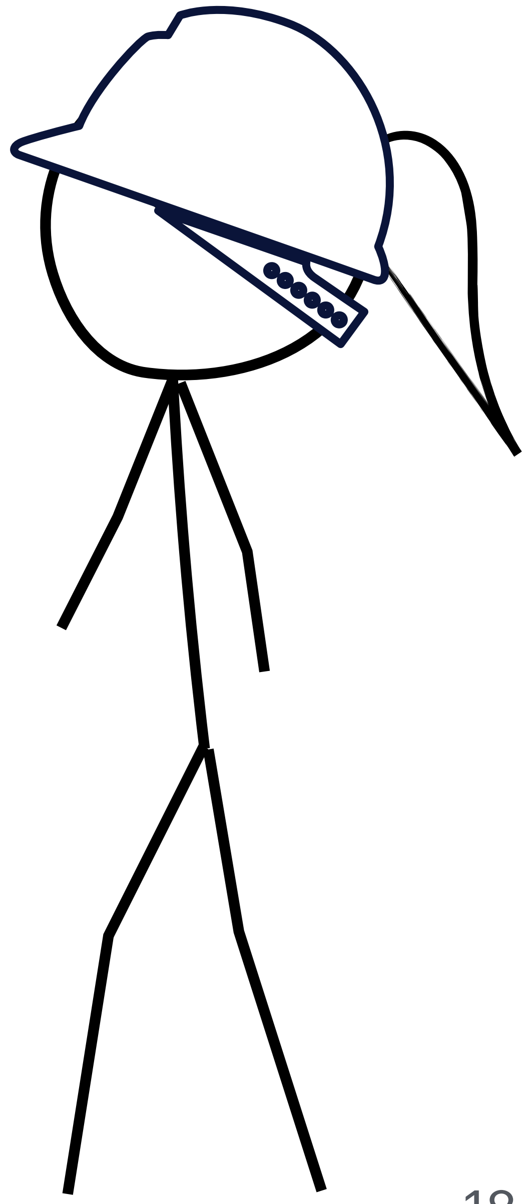


```
type proof = string list

module Prover : AUTHENTIKIT =
  type 'a auth = 'a * string
  type 'a auth_computation = () -> proof * 'a

  let return a () = ([], a)
  let bind c f =
    let (prf,  a) = c () in
    let (prf', b) = f a () in
    (prf @ prf', b)

  module Serializable = struct
    type 'a evidence = 'a -> string

    (* ... *)
  end

  let auth evi a = (a, hash (evi a))
  let unauth evi (a, _) () = ([evi a], a)
end
```

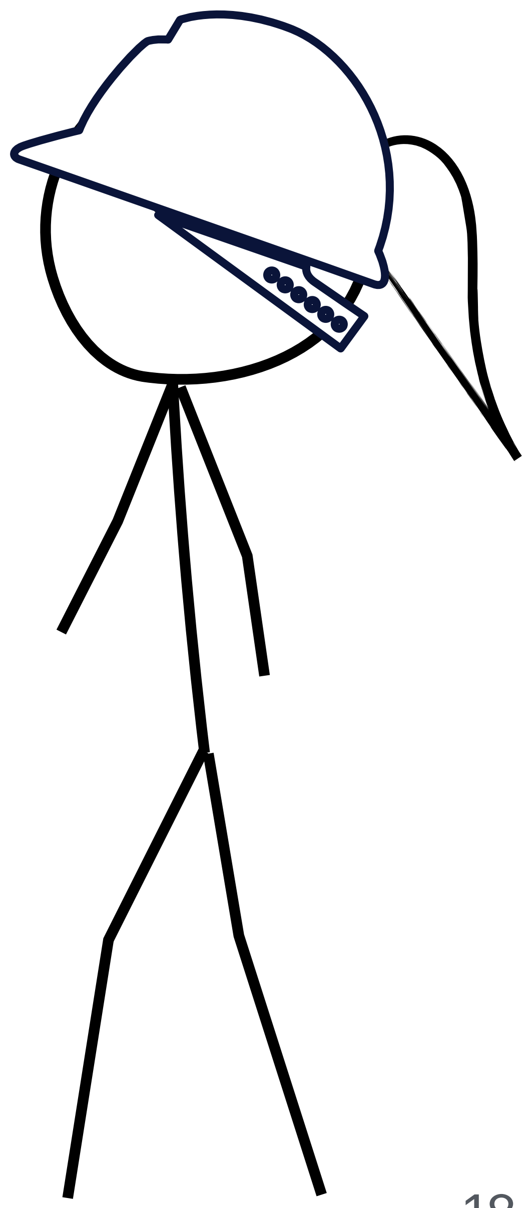
```
module Verifier : AUTHENTIKIT =
  type 'a auth = string
  type 'a auth_computation =
    proof -> [`Ok of proof * 'a | `ProofFailure]







  (* ... *)









  (* ... *)




end
```

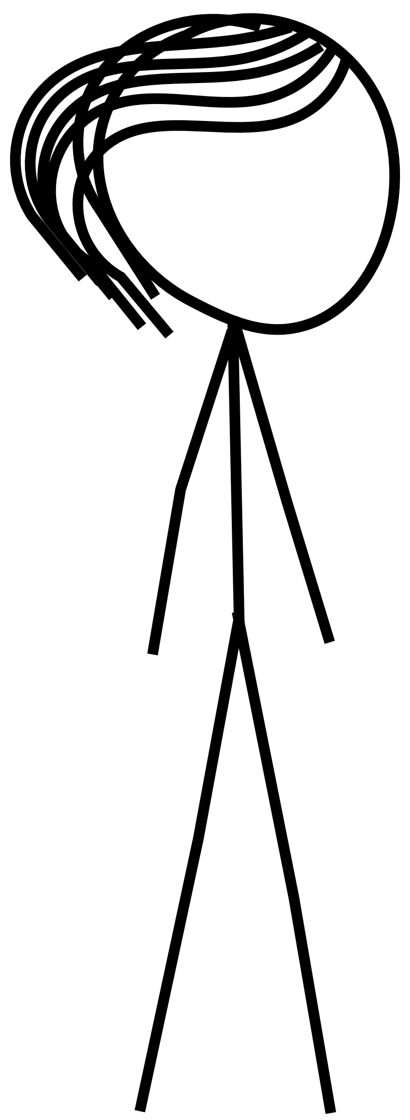


```
module Verifier : AUTHENTIKIT =
  type 'a auth = string
  type 'a auth_computation =
    proof -> [`Ok of proof * 'a | `ProofFailure]

  let return a prf = `Ok (prf, a)
  let bind c f prf =
    match c prf with
    | `ProofFailure -> `ProofFailure
    | `Ok (prf', a) -> f a prf'

  module Serializable = struct
    type 'a evidence =
      { serialize : 'a -> string; deserialize : string -> 'a option }

    (* ... *)
  end




   (* ... *)




end
```



```
module Verifier : AUTHENTIKIT =
  type 'a auth = string
  type 'a auth_computation =
    proof -> [`Ok of proof * 'a | `ProofFailure]

  let return a prf = `Ok (prf, a)
  let bind c f prf =
    match c prf with
    | `ProofFailure -> `ProofFailure
    | `Ok (prf', a) -> f a prf'

  module Serializable = struct
    type 'a evidence =
      { serialize : 'a -> string; deserialize : string -> 'a option }

    (* ... *)
  end

  let auth evi a = hash (evi.serialize a)
  let unauth evi h prf =
    match prf with
    | p :: ps when hash p = h ->
      match evi.deserialize p with
      | None   -> 'ProofFailure
      | Some a -> `Ok (ps, a)
    | _ -> 'ProofFailure
end
```





```
module Ideal : AUTHENTIKIT = struct
  type 'a auth = 'a
  type 'a auth_computation = () -> 'a

  let return a () = a
  let bind a f () = f (a ()) ()

  (* ... *)

  let auth _ a = a
  let unauth _ a () = a
end
```

Takeaway

Takeaway

- In the end, it is not so difficult to prove that **one particular client** has the security and correctness property.
- The challenge is to prove that **any well-typed client** has these properties!
- Authentikit relies on a **parametricity** property of OCaml's module system. In fact, we prove security and correctness as "free" theorems.
- To do this, we define two logical relations.

Requirements

```
module type AUTHENTIKIT = sig
  type 'a auth
  type 'a auth_computation

  val return : 'a -> 'a auth_computation
  val bind   : 'a auth_computation -> (‘a -> 'b auth_computation) -> ‘b auth_computation

  module Serializable : sig
    type 'a evidence


    (* ... *)


  end

  val auth   : 'a Serializable.evidence -> 'a -> 'a auth
  val unauth : 'a Serializable.evidence -> 'a auth -> 'a auth_computation
end
```

Requirements

```
module type AUTHENTIKIT = sig
  type 'a auth
  type 'a auth_computation

  val return : 'a -> 'a auth_computation
  val bind   : 'a auth_computation -> (‘a -> 'b auth_computation) -> ‘b auth_computation

  module Serializable : sig
    type 'a evidence


    (* ... *)


  end

  val auth   : 'a Serializable.evidence -> 'a -> 'a auth
  val unauth : 'a Serializable.evidence -> 'a auth -> 'a auth_computation
end
```

(higher-order) functions

Requirements

```
module type AUTHENTIKIT = sig
  type 'a auth
  type 'a auth_computation

  val return : 'a -> 'a auth_computation
  val bind   : 'a auth_computation -> (‘a -> 'b auth_computation) -> ‘b auth_computation

  module Serializable : sig
    type 'a evidence


    (* ... *)


  end

  val auth   : 'a Serializable.evidence -> 'a -> 'a auth
  val unauth : 'a Serializable.evidence -> 'a auth -> 'a auth_computation
end
```

(higher-order) functions

polymorphism

Requirements

```
module Merkle : MERKLE = functor (A : AUTHENTIKIT) -> struct
  open A

  type path = [`L | `R] list
  type tree = [`leaf of string | `node of tree auth * tree auth]

  (* ... *)

end
```

recursive types

```
module type AUTHENTIKIT = sig
  type 'a auth
  type 'a auth_computation

  val return : 'a -> 'a auth_computation
  val bind   : 'a auth_computation -> (‘a -> 'b auth_computation) -> ‘b auth_computation

  module Serializable : sig
    type 'a evidence


    (* ... *)


  end

  val auth   : 'a Serializable.evidence -> 'a -> 'a auth
  val unauth : 'a Serializable.evidence -> 'a auth -> 'a auth_computation
end
```

(higher-order) functions

polymorphism

Requirements

```
module Merkle : MERKLE = functor (A : AUTHENTIKIT) -> struct
  open A

  type path = [`L | `R] list
  type tree = [`leaf of string | `node of tree auth * tree auth]

  (* ... *)

end
```

recursive types

```
module type AUTHENTIKIT = sig
  type 'a auth
  type 'a auth_computation

  val return : 'a -> 'a auth_computation
  val bind   : 'a auth_computation -> (‘a -> 'b auth_computation) -> ‘b auth_computation

  module Serializable : sig
    type 'a evidence


    (* ... *)


  end

  val auth   : 'a Serializable.evidence -> 'a -> 'a auth
  val unauth : 'a Serializable.evidence -> 'a auth -> 'a auth_computation
end
```

(higher-order) functions

state

polymorphism

```
module Prover : AUTHENTIKIT
```

Requirements

```
module Merkle : MERKLE = functor (A : AUTHENTIKIT) -> struct
  open A

  type path = [`L | `R] list
  type tree = [`leaf of string | `node of tree auth * tree auth]

  (* ... *)

end
```

recursive types

```
module type AUTHENTIKIT = sig
  type 'a auth
  type 'a auth_computation

  val return : 'a -> 'a auth_computation
  val bind   : 'a auth_computation -> ('a -> 'b auth_computation) -> 'b auth_computation

  module Serializable : sig
    type 'a evidence


    (* ... *)


  end

  val auth   : 'a Serializable.evidence -> 'a -> 'a auth
  val unauth : 'a Serializable.evidence -> 'a auth -> 'a auth_computation
end
```

abstract type constructors

(higher-order) functions

polymorphism

state

```
module Prover : AUTHENTIKIT
```

The $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$ language

$$\kappa ::= \star \mid \kappa \Rightarrow \kappa \qquad \text{(kinds)}$$

$$\tau ::= \alpha \mid \lambda\alpha : \kappa.\tau \mid \tau\ \tau \mid c \qquad \text{(types)}$$

$$c ::= \ldots \mid \times \mid + \mid \rightarrow \mid \mathsf{ref} \mid \forall_\kappa \mid \exists_\kappa \mid \mu_\kappa \qquad \text{(constructors)}$$

$$\frac{\Theta \vdash \tau \equiv \sigma \qquad \Theta \mid \Gamma \vdash e : \sigma}{\Theta \mid \Gamma \vdash e : \tau} \qquad \frac{}{\Theta \vdash (\lambda\alpha.\tau)\sigma \equiv \tau[\sigma/\alpha]}$$

Authentikit in $F_{\omega,\mu}^{\text{ref}}$

```
module type AUTHENTIKIT = sig
  type 'a auth
  type 'a auth_computation

  val return : 'a -> 'a auth_computation
  val bind   : 'a auth_computation ->
               (‘a -> 'b auth_computation) ->
               'b auth_computation

  module Serializable : sig
    type 'a evidence

     (* ... *)

  end

  val auth   : 'a Serializable.evidence -> 'a -> 'a auth
  val unauth : 'a Serializable.evidence ->
               ‘a auth -> 'a auth_computation
end
```

$$\mathsf{AUTHENTIKIT} \triangleq \exists \mathsf{auth}, \mathsf{m} : \star \Rightarrow \star.\ \mathsf{Authentikit}\ \mathsf{auth}\ \mathsf{m}$$

$$\begin{aligned}\mathsf{Authentikit} \triangleq\ & \lambda \mathsf{auth}, \mathsf{m} : \star \Rightarrow \star. \\ & (\forall \alpha : \star.\ \alpha \to \mathsf{m}\ \alpha) \times \\ & (\forall \alpha, \beta : \star.\ \mathsf{m}\ \alpha \to (\alpha \to \mathsf{m}\ \beta) \to \mathsf{m}\ \beta) \times \\ & \quad \vdots \\ & (\forall \alpha : \star.\ \mathsf{evidence}\ \alpha \to \alpha \to \mathsf{auth}\ \alpha) \times \\ & (\forall \alpha : \star.\ \mathsf{evidence}\ \alpha \to \mathsf{auth}\ \alpha \to \mathsf{m}\ \alpha)\end{aligned}$$

“F-ing” the module

Our approach

Our approach

To show security and correctness we

Our approach

To show security and correctness we

1. Define **Collision-Free Separation Logic** (CFSL).

Our approach

To show security and correctness we

1. Define **Collision-Free Separation Logic** (CFSL).

2. Define **binary** and **ternary logical relations** for security and correctness.

Our approach

To show security and correctness we

1. Define **Collision-Free Separation Logic** (CFSL).

2. Define **binary** and **ternary logical relations** for security and correctness.

3. Show security and correctness as free theorems by verifying implementations of the **Prover**, **Verifier**, and **Ideal** semantically inhabit the Authentikit type.

Theorem (Security)

If e is a program parameterized by an Authentikit implementation, i.e.,

Theorem (Security)
If e is a program parameterized by an Authentikit implementation, i.e.,

$$\vdash e : \forall \mathsf{auth}, \mathsf{m}.\ \mathsf{Authentikit}\ \mathsf{auth}\ \mathsf{m} \rightarrow \mathsf{m}\ \tau$$

Theorem (Security)
If e is a program parameterized by an Authentikit implementation, i.e.,

$$\vdash e : \forall \mathsf{auth}, \mathsf{m}.\ \mathsf{Authentikit\ auth\ m} \rightarrow \mathsf{m}\ \tau$$

then for all proofs p, if

Theorem (Security)

If e is a program parameterized by an Authentikit implementation, i.e.,

$$\vdash e : \forall \mathsf{auth}, \mathsf{m}.\ \mathsf{Authentikit\ auth\ m} \rightarrow \mathsf{m}\ \tau$$

then for all proofs p, if

e instantiated with **Verifier** accepts p and returns v

Theorem (Security)
If e is a program parameterized by an Authentikit implementation, i.e.,

$$\vdash e : \forall \mathsf{auth}, \mathsf{m}.\ \mathsf{Authentikit\ auth\ m} \to \mathsf{m}\ \tau$$

then for all proofs p, if

e instantiated with **Verifier** accepts p and returns v

then

Theorem (Security)
If e is a program parameterized by an Authentikit implementation, i.e.,

$$\vdash e : \forall \mathsf{auth}, \mathsf{m}.\ \mathsf{Authentikit\ auth\ m} \rightarrow \mathsf{m}\ \tau$$

then for all proofs p, if

e instantiated with **Verifier** accepts p and returns v

then

- e instantiated with **Ideal** returns v or

Theorem (Security)
If e is a program parameterized by an Authentikit implementation, i.e.,

$$\vdash e : \forall \mathrm{auth}, \mathrm{m}.\ \mathrm{Authentikit\ auth\ m} \rightarrow \mathrm{m}\ \tau$$

then for all proofs p, if

e instantiated with **Verifier** accepts p and returns v

then

- e instantiated with **Ideal** returns v or
- a **hash collision** occurred

Theorem (Correctness)
If e is a program parameterized by an Authentikit implementation, i.e.,

$$\vdash e : \forall \mathsf{auth}, \mathsf{m}.\ \mathsf{Authentikit\ auth\ m} \rightarrow \mathsf{m}\ \tau$$

then if

e instantiated with **Prover** produces a proof p and returns v

then

- e instantiated with **Verifier** accepts p and returns v and
- e instantiated with **Ideal** returns v as well.

Theorem (Correctness)
If e is a program parameterized by an Authentikit implementation, i.e.,

$$\vdash e : \forall \mathsf{auth}, \mathsf{m}.\ \mathsf{Authentikit\ auth\ m} \to \mathsf{m}\ \tau$$

then if

e instantiated with **Prover** produces a proof p and returns v

then

- e instantiated with **Verifier** accepts p and returns v and
- e instantiated with **Ideal** returns v as well.

Collision-free reasoning

To define our models, we define **Collision-Free Separation Logic** (CF-SL),

$$\text{wp } e \ \{\Phi\}$$

that is expressive enough to state and prove security and correctness.

CF-SL statements hold “up to” hash collision.

CF-SL

CF-SL satisfies all the standard weakest precondition rules but introduces a resource $\text{hashed}(s)$ such that

$$\frac{}{\text{wp } \mathsf{hash}\ s\ \{v.\, v = H(s) * \text{hashed}(s)\}}$$

and

$$\frac{\mathit{collision}(s_1, s_2)}{\text{hashed}(s_1) * \text{hashed}(s_2) \vdash \text{False}}$$

Interpreting $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$

Kinds:

$$[\![\star]\!] \triangleq Val \times Val \to iProp_{\Box}$$

$$[\![\kappa_1 \Rightarrow \kappa_2]\!] \triangleq [\![\kappa_1]\!] \xrightarrow{\mathsf{ne}} [\![\kappa_2]\!]$$

Interpreting $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$

Kinds:

$$[\![\star]\!] \triangleq Val \times Val \rightarrow \mathit{iProp}_{\Box}$$

$$[\![\kappa_1 \Rightarrow \kappa_2]\!] \triangleq [\![\kappa_1]\!] \xrightarrow{\mathsf{ne}} [\![\kappa_2]\!]$$

Types:

$$[\![\Theta \vdash \tau : \kappa]\!]_{\Delta} \;:\; [\![\kappa]\!]$$

Interpreting $\mathsf{F}_{\omega,\mu}^{\mathsf{ref}}$

Kinds:

$$[\![\star]\!] \triangleq \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_{\Box}$$

$$[\![\kappa_1 \Rightarrow \kappa_2]\!] \triangleq [\![\kappa_1]\!] \xrightarrow{\mathsf{ne}} [\![\kappa_2]\!]$$

Types:

$$[\![\Theta \vdash \tau : \kappa]\!]_\Delta \ : \ [\![\kappa]\!]$$

$$[\![\Theta \vdash \alpha : \kappa]\!]_\Delta \triangleq \Delta(\alpha)$$

Interpreting $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$

Kinds:

$$[\![\star]\!] \triangleq \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_{\Box}$$

$$[\![\kappa_1 \Rightarrow \kappa_2]\!] \triangleq [\![\kappa_1]\!] \xrightarrow{\mathsf{ne}} [\![\kappa_2]\!]$$

Types:

$$[\![\Theta \vdash \tau : \kappa]\!]_\Delta \quad : \quad [\![\kappa]\!]$$

$$[\![\Theta \vdash \alpha : \kappa]\!]_\Delta \triangleq \Delta(\alpha)$$

$$[\![\Theta \vdash \lambda\alpha.\, \tau : \kappa_1 \Rightarrow \kappa_2]\!]_\Delta \triangleq \lambda R : [\![\kappa_1]\!].\, [\![\Theta, \alpha : \kappa_1 \vdash \tau : \kappa_2]\!]_{\Delta,R}$$

Interpreting $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$

Kinds:

$$[\![\star]\!] \triangleq \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_{\square}$$

$$[\![\kappa_1 \Rightarrow \kappa_2]\!] \triangleq [\![\kappa_1]\!] \xrightarrow{\mathsf{ne}} [\![\kappa_2]\!]$$

Types:

$$[\![\Theta \vdash \tau : \kappa]\!]_\Delta \;:\; [\![\kappa]\!]$$

$$[\![\Theta \vdash \alpha : \kappa]\!]_\Delta \triangleq \Delta(\alpha)$$

$$[\![\Theta \vdash \lambda\alpha.\, \tau : \kappa_1 \Rightarrow \kappa_2]\!]_\Delta \triangleq \lambda R : [\![\kappa_1]\!].\, [\![\Theta, \alpha : \kappa_1 \vdash \tau : \kappa_2]\!]_{\Delta,R}$$

$$[\![\Theta \vdash \sigma\ \tau : \kappa_2]\!]_\Delta \triangleq [\![\Theta \vdash \sigma : \kappa_1 \Rightarrow \kappa_2]\!]_\Delta\,([\![\Theta \vdash \tau : \kappa_1]\!]_\Delta)$$

Interpreting $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$

Kinds:

$$[\![\star]\!] \triangleq \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_{\Box}$$

$$[\![\kappa_1 \Rightarrow \kappa_2]\!] \triangleq [\![\kappa_1]\!] \xrightarrow{\mathsf{ne}} [\![\kappa_2]\!]$$

Types:

$$[\![\Theta \vdash \tau : \kappa]\!]_\Delta \;:\; [\![\kappa]\!]$$

$$[\![\Theta \vdash \alpha : \kappa]\!]_\Delta \triangleq \Delta(\alpha)$$

$$[\![\Theta \vdash \lambda\alpha.\,\tau : \kappa_1 \Rightarrow \kappa_2]\!]_\Delta \triangleq \lambda R : [\![\kappa_1]\!].\, [\![\Theta, \alpha : \kappa_1 \vdash \tau : \kappa_2]\!]_{\Delta,R}$$

$$[\![\Theta \vdash \sigma\;\tau : \kappa_2]\!]_\Delta \triangleq [\![\Theta \vdash \sigma : \kappa_1 \Rightarrow \kappa_2]\!]_\Delta\,([\![\Theta \vdash \tau : \kappa_1]\!]_\Delta)$$

$$[\![\Theta \vdash c : \kappa]\!]_\Delta \triangleq [\![c : \kappa]\!]$$

Interpreting $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$

Constructors:

$$[\![\mathsf{bool} : \star]\!] \triangleq \lambda(v_1, v_2).\ \exists b \in \mathbb{B}.\ v_1 = v_2 = b$$

Interpreting $\mathsf{F}^{\mathsf{ref}}_{\omega,\mu}$

Constructors:

$$[\![\mathsf{bool} : \star]\!] \triangleq \lambda(v_1, v_2).\, \exists b \in \mathbb{B}.\, v_1 = v_2 = b$$

$$\vdots$$

$$\begin{aligned}[\![\times : \star \Rightarrow \star \Rightarrow \star]\!] \triangleq \lambda R, S : [\![\star]\!].\, \lambda(v_1, v_2).\, \exists w_1, w_2, u_1, u_2.\\ v_1 = (w_1, u_1) * v_2 = (w_2, u_2) * R(w_1, w_2) * S(u_1, u_2)\end{aligned}$$

Verifying Authentikit implementations

Verifying Authentikit implementations

Security

$$\llbracket \mathsf{auth} \rrbracket \triangleq \lambda A, (v_1, v_2).\, \exists a, t.\, v_1 = H(serialize_t(a)) * A(a, v_2) * \mathsf{hashed}(serialize_t(a))$$

$$\llbracket \mathsf{m} \rrbracket \triangleq \lambda A, (v_1, v_2).\, \forall p.\, \{\mathsf{isProof}(p)\}\ v_1\ p \sim v_2\ ()\ \{Q_{\mathsf{post}}\}$$

Verifying Authentikit implementations

Security

$$[\![\mathsf{auth}]\!] \triangleq \lambda A, (v_1, v_2).\, \exists a, t.\, v_1 = H(serialize_t(a)) * A(a, v_2) * \mathsf{hashed}(serialize_t(a))$$

$$[\![\mathsf{m}]\!] \triangleq \lambda A, (v_1, v_2).\, \forall p.\, \{\mathsf{isProof}(p)\}\ v_1\ p \sim v_2\ ()\ \{Q_{\mathsf{post}}\}$$

Correctness

$$[\![\mathsf{m}]\!] \triangleq \lambda A, (v_1, v_2, v_3).\, \forall p.\, \{\mathsf{isProphProof}(p)\}\ v_1\ () \sim v_2\ p \sim v_3\ ()\ \{Q'_{\mathsf{post}}\}$$

Verifying Authentikit implementations

Security

$$[\![\mathsf{auth}]\!] \triangleq \lambda A, (v_1, v_2).\, \exists a, t.\, v_1 = H(\mathit{serialize}_t(a)) \ast$$

$$[\![\mathsf{m}]\!] \triangleq \lambda A, (v_1, v_2).\, \forall p.\, \{\mathsf{isProof}(p)\}\ v_1\ p \sim v_2\ ()$$

```
module Prover : AUTHENTIKIT =
  (* ... *)

  let unauth evi (a, _) p () =
    let s = evi a in
    resolve p to s;
    ([s], a)
end
```

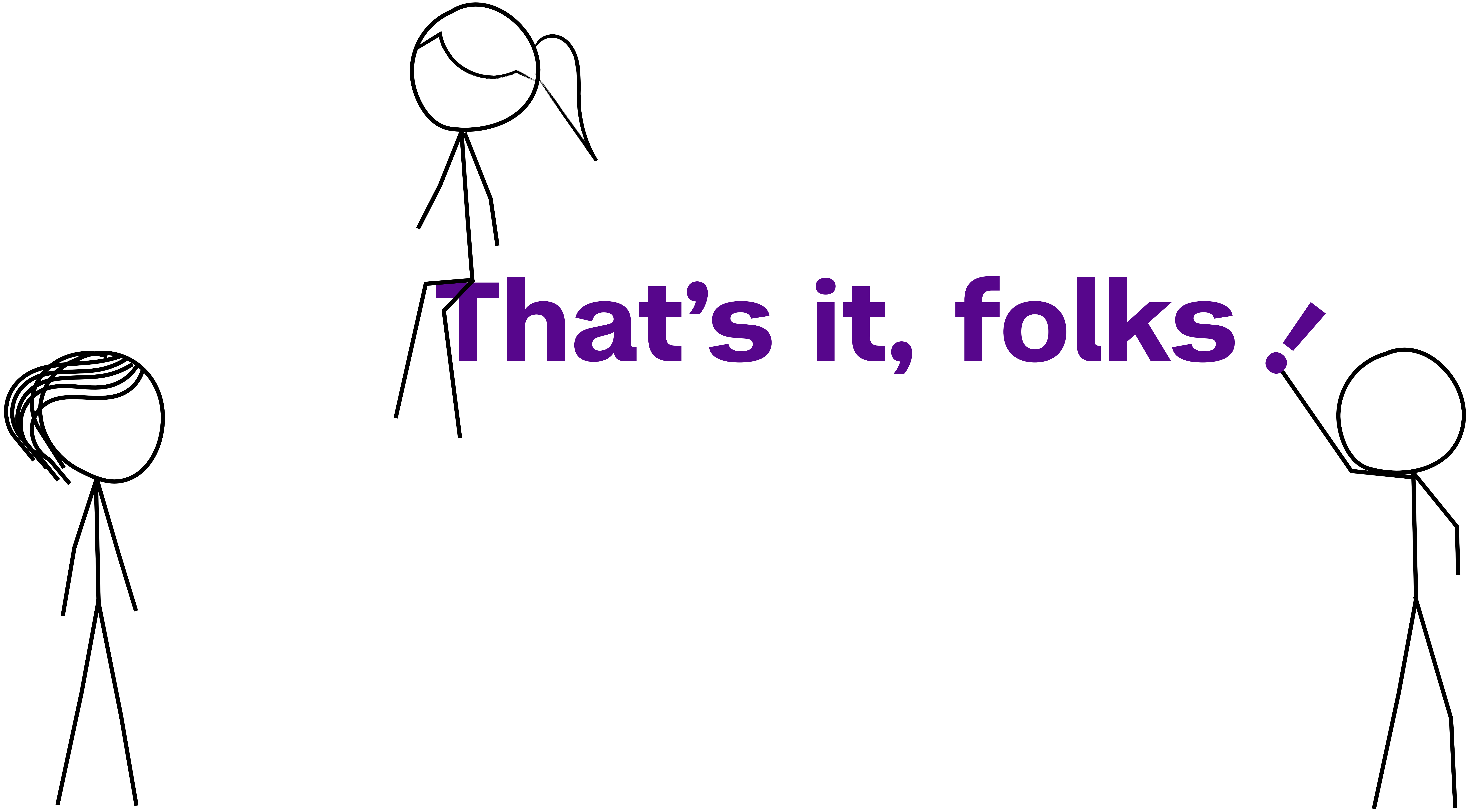
Correctness

$$[\![\mathsf{m}]\!] \triangleq \lambda A, (v_1, v_2, v_3).\, \forall p.\, \{\mathsf{isProphProof}(p)\}\ v_1\ () \sim v_2\ p \sim v_3\ ()\ \{Q'_{\mathsf{post}}\}$$

Summary

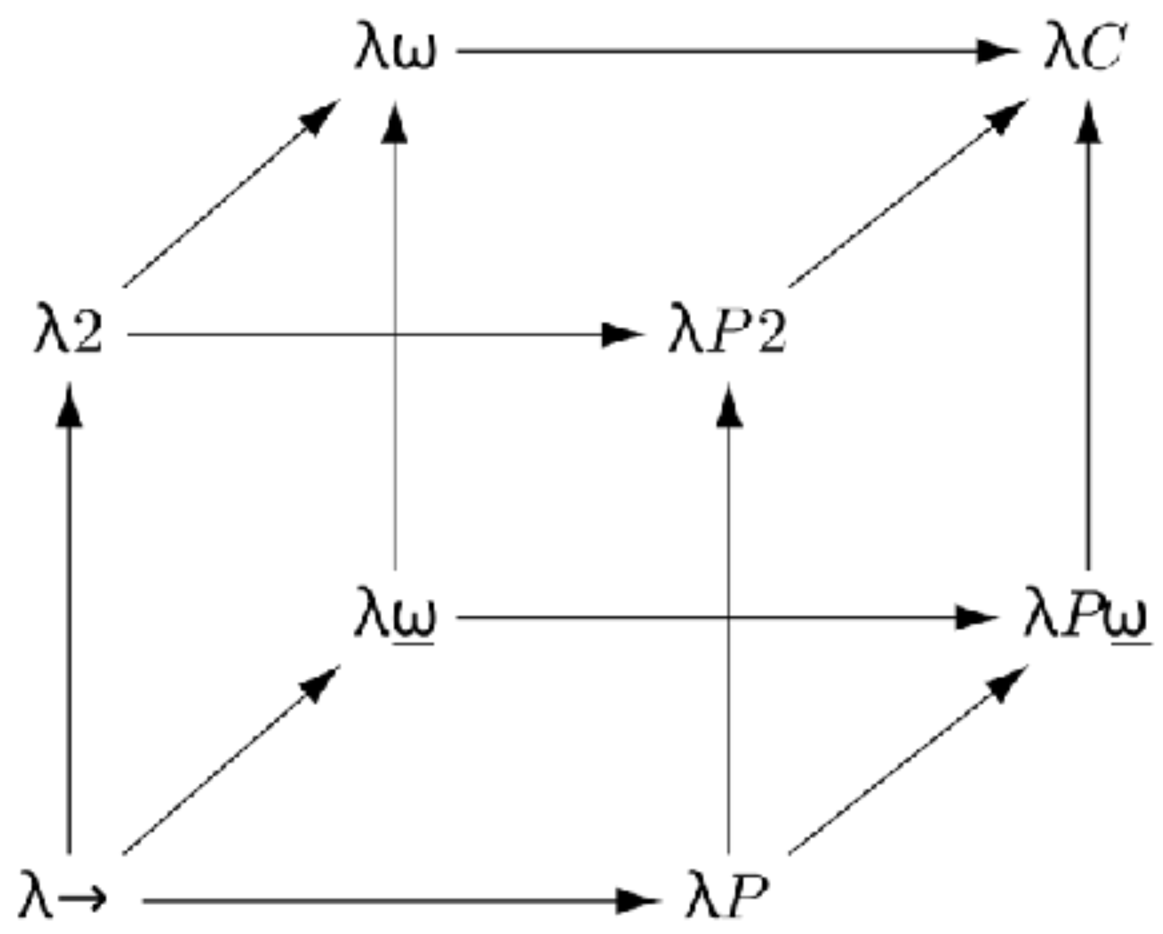
- **Authentikit** is a library for implementing ADSs generically.
- Two **logical-relations models** and a proof of security and correctness of the Authentikit module functor construction in OCaml.
  - We verify several **optimizations**.
  - We show how to **safely link** manually verified code with code automatically generated using Authentikit by semantic typing.
- Full mechanization in the Rocq theorem prover.

https://arxiv.org/abs/2501.10802


That's it, folks!

```ocaml
module type AUTHENTIKIT = sig
  type 'a auth
  type 'a auth_computation

  val return : 'a -> 'a auth_computation
  val bind   : 'a auth_computation -> ('a -> 'b auth_computation) -> 'b auth_computation

  module Serializable : sig
    type 'a evidence
    val auth   : 'a auth evidence
    val pair   : 'a evidence -> 'b evidence -> ('a * 'b) evidence
    val sum    : 'a evidence -> 'b evidence -> [`left of 'a | `right of 'b] evidence
    val string : string evidence
    val int    : int evidence
  end

  val auth   : 'a Serializable.evidence -> 'a -> 'a auth
  val unauth : 'a Serializable.evidence -> 'a auth -> 'a auth_computation
end
```

Reminder



STLC: terms can depend on terms,

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x \,.\, e : \sigma \to \tau}$$

System F: terms can depend on types,

$$\frac{\Theta, \alpha \mid \Gamma \vdash e : \tau}{\Theta \mid \Gamma \vdash \Lambda \alpha \,.\, e : \forall \alpha \,.\, \tau}$$

System F_{ω}: types can depend on types,

$$\frac{\Theta \vdash \tau \equiv \sigma \qquad \Theta \mid \Gamma \vdash e : \sigma}{\Theta \mid \Gamma \vdash e : \tau} \qquad\qquad \frac{}{\Theta \vdash (\lambda \alpha \,.\, \tau)\sigma \equiv \tau[\sigma/\alpha]}$$

Security

To show security of Authentikit, we use CF-SL to define a **logical relation**

$$\Theta \mid \Gamma \vDash e_1 \sim e_2 : \tau$$

and show

1. If $\Theta \mid \Gamma \vdash e : \tau$ then $\Theta \mid \Gamma \vDash e \sim e : \tau$

2. If $\Theta \mid \Gamma \vDash e_1 \sim e_2 : \tau$ then e_1 and e_2 are secure (as verifier and ideal)

3. $\varnothing \mid \varnothing \vDash \text{Authentikit}_V \sim \text{Authentikit}_I : \text{AUTHENTIKIT}$

Logical relation, sketch

Intuitively, the judgment $\varnothing \mid \varnothing \vDash e_1 \sim e_2 : \tau$ means

$$\{\mathsf{True}\}\, e_1 \sim e_2 \,\{[\![\tau]\!]\}$$

where $[\![\tau]\!] : \mathsf{Val} \times \mathsf{Val} \to \mathsf{iProp}$ is an **interpretation of types**. E.g.

$$[\![\mathbb{N}]\!](v_1, v_2) \triangleq \exists n \in \mathbb{N}\,.\, v_1 = v_2 = n$$
$$[\![\tau_1 \to \tau_2]\!](v_1, v_2) \triangleq \forall w_1, w_2\,.\, \{[\![\tau_1]\!](w_1, w_2)\}\, v_1\ w_1 \sim v_2\ w_2 \,\{[\![\tau_2]\!]\}$$

Security proof

The main work is to show

$$[\![\text{Authentikit auth m}]\!](\text{Authentikit}_V, \text{Authentikit}_I)$$

The challenging part is finding the right interpretation of the type variables.

$$\begin{aligned}
[\![\text{auth}]\!](A)(v_1, v_2) &\triangleq \exists a, t.\, v_1 = \mathit{hash}(\mathit{serialize}_t(a)) * A(a, v_2) * \text{hashed}(\mathit{serialize}_t(a)) \\
[\![\text{m}]\!](A)(v_1, v_2) &\triangleq \forall p.\, \{\text{isProof}(p)\}\ v_1\ p \sim v_2\ ()\ \{Q_{\text{post}}\} \\
Q_{\text{post}}(u_1, u_2) &\triangleq u_1 = \text{None} \vee (\exists a_1, p'.\, u_1 = \text{Some}(p', a_1) * \text{isProof}(\text{p}') * A(a_1, u_2))
\end{aligned}$$

Optimizations of Authentikit

- Proof accumulator
- Proof-reuse buffering
- Heterogeneous buffering
- Stateful buffering

```
module Verifier : AUTHENTIKIT =
  type 'a auth_computation =
    pfstate -> [`Ok of pfstate * 'a | `ProofFailure]

  (* ... *)

  let unauth evi h pf =
    match Map.find_opt h pf.cache with
    | None ->
       match pf.pf_stream with
       | [] -> `ProofFailure
       | p :: ps when hash p = h ->
          match evi.deserialize p with
          | None -> `ProofFailure
          | Some a ->
           `Ok ({pf_stream = ps;
                 cache = Map.add h p pf.cache}, a)
       | _ -> `ProofFailure
    | Some p ->
       match evi.deserialize p with
       | None -> `ProofFailure
       | Some a -> `Ok (pf, a)

end
```

Manual client proofs

The naïve implementation of Authentikit does not emit optimal proofs, e.g.,

$$\text{lookup}([R, L], t_0) = ([(h_1, h_2), (h_5, h_6), s_5], s_5)$$

Instead, we can manually implement and "semantically type" the optimal strategy:

$$[\![\, \text{path} \to \text{auth tree} \to \text{m (option string)} \,]\!](\text{fetch}_V, \text{fetch}_I)$$