NYU

Logical Relations for Formally Verified

Authenticated Data Structures

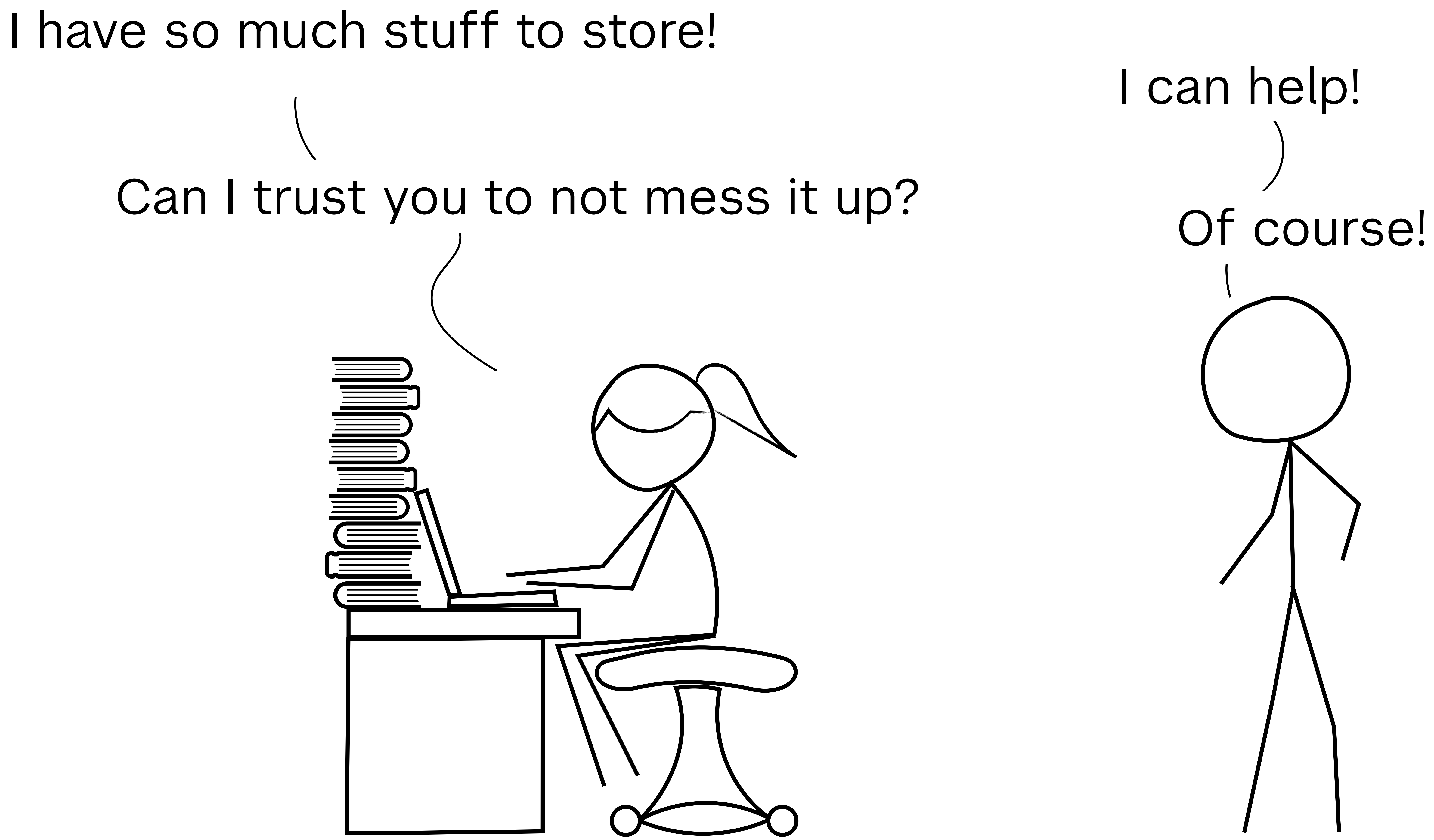
Simon Oddershede Gregersen
joint work with Chaitanya Agarwal and Joseph Tassarotti

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 outsource data storage and processing to Bob?

How can Alice outsource data storage and processing to Bob?

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

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

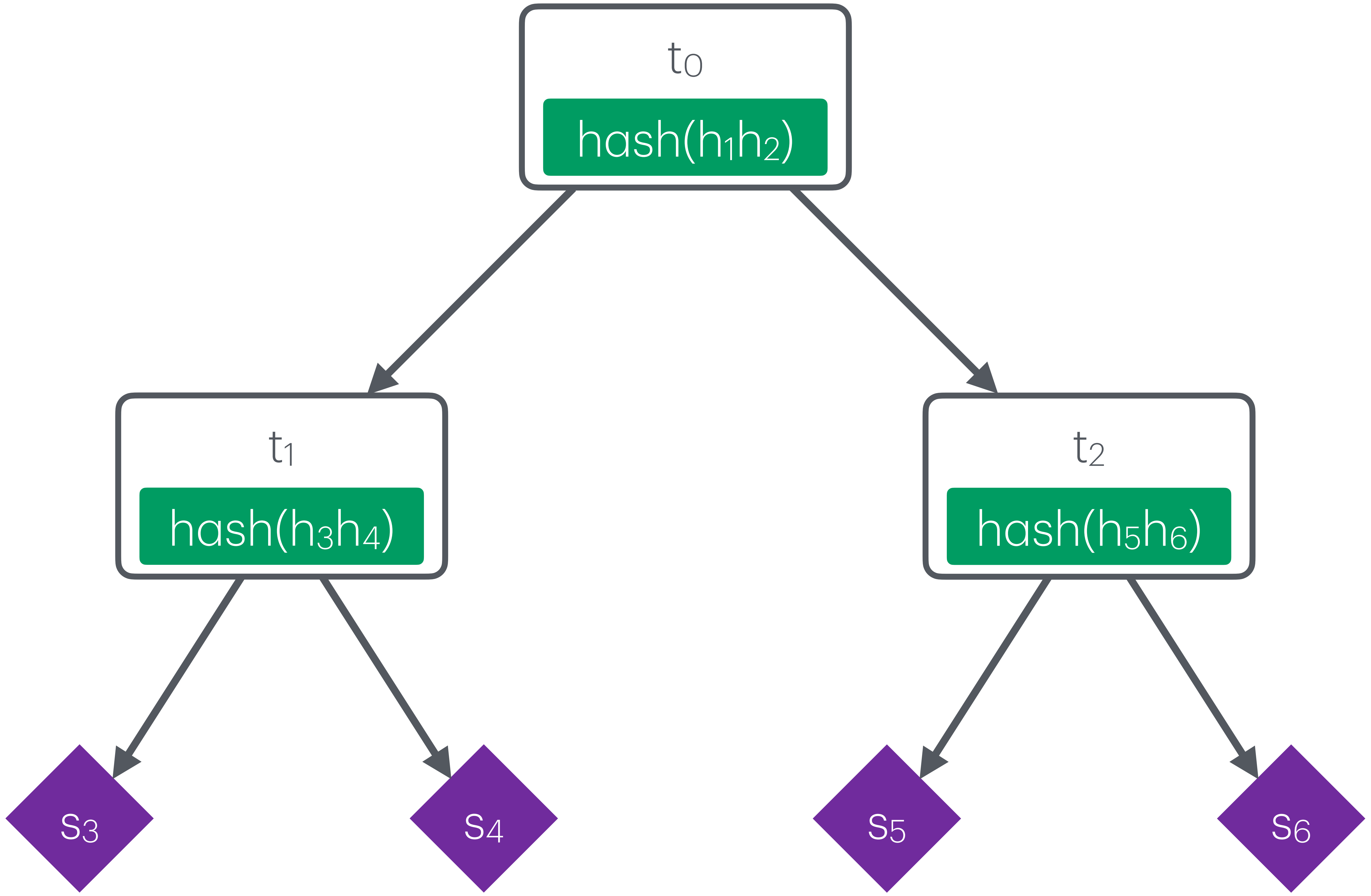
How can Alice outsource data storage and processing to Bob?

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

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

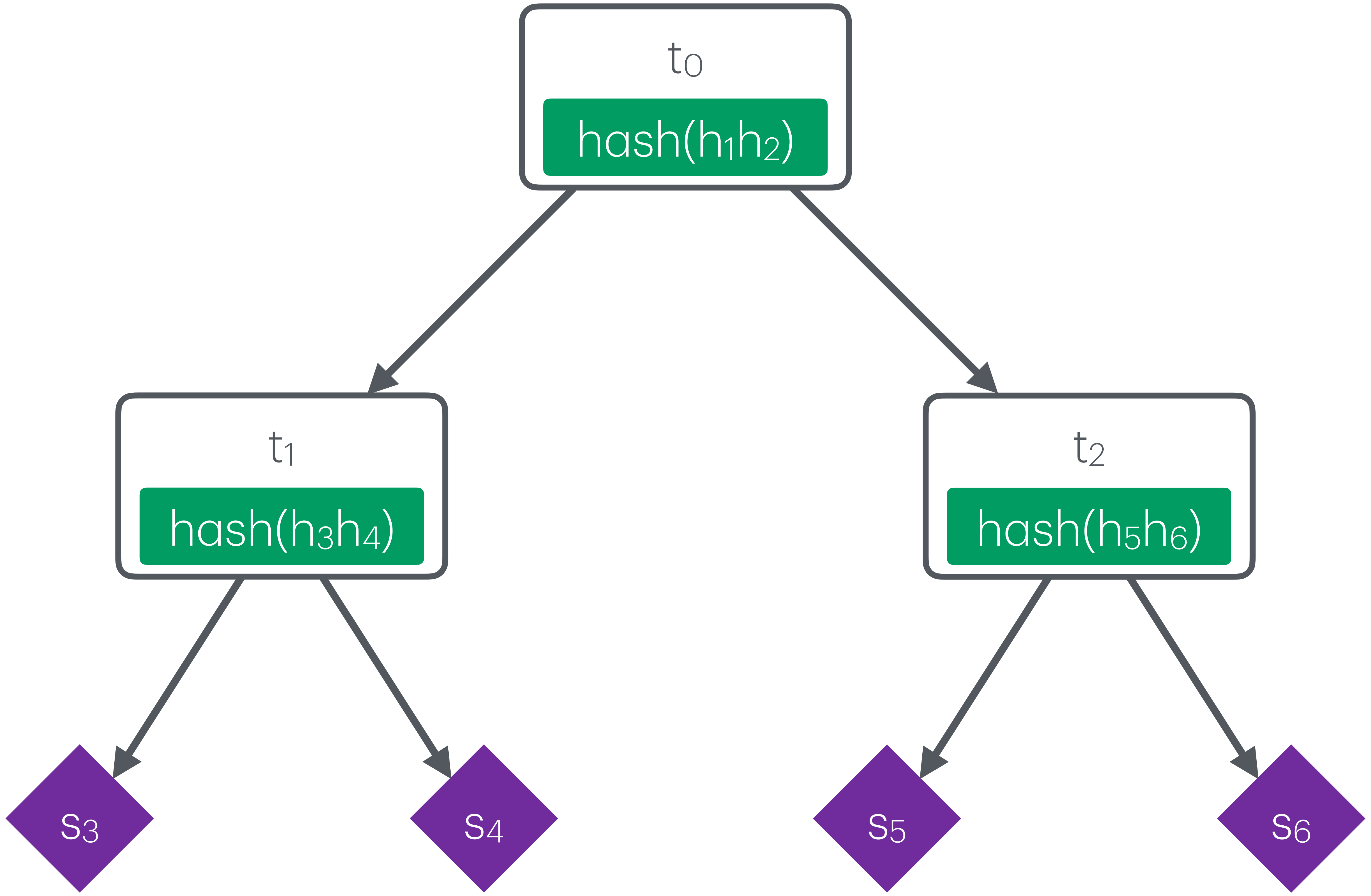
ADSs allow outsourcing data storage and processing tasks to untrusted servers without loss of integrity.

Example: Merkle Tree

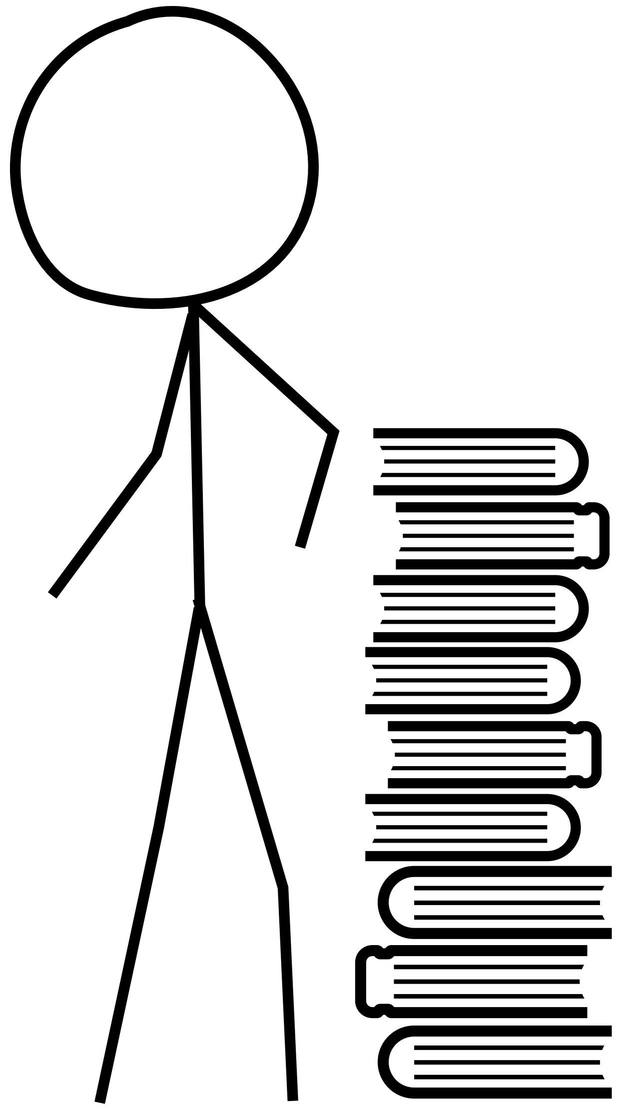


where h_i denotes the hash of t_i / s_i

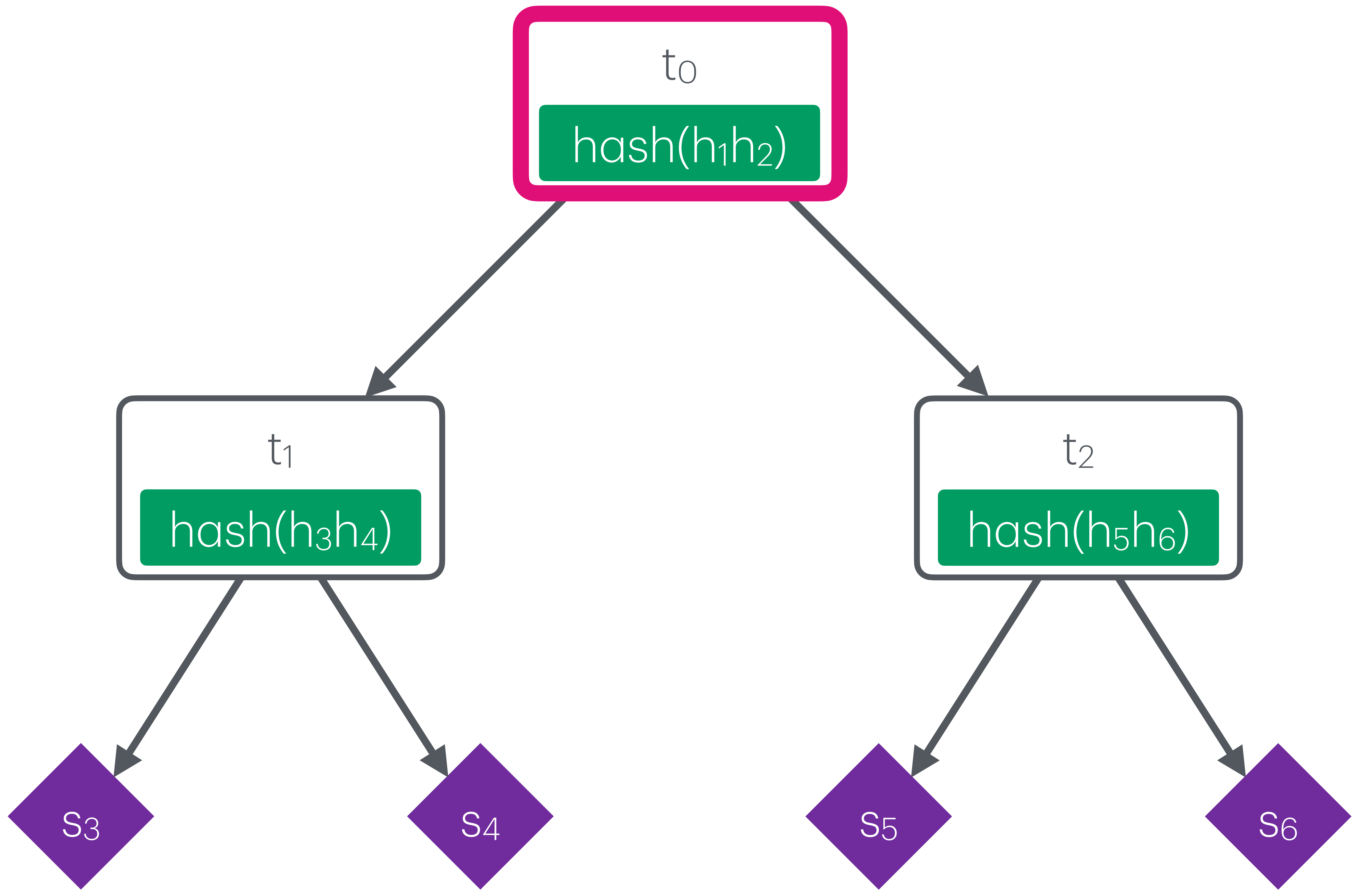
Example: Merkle Tree (Prover)



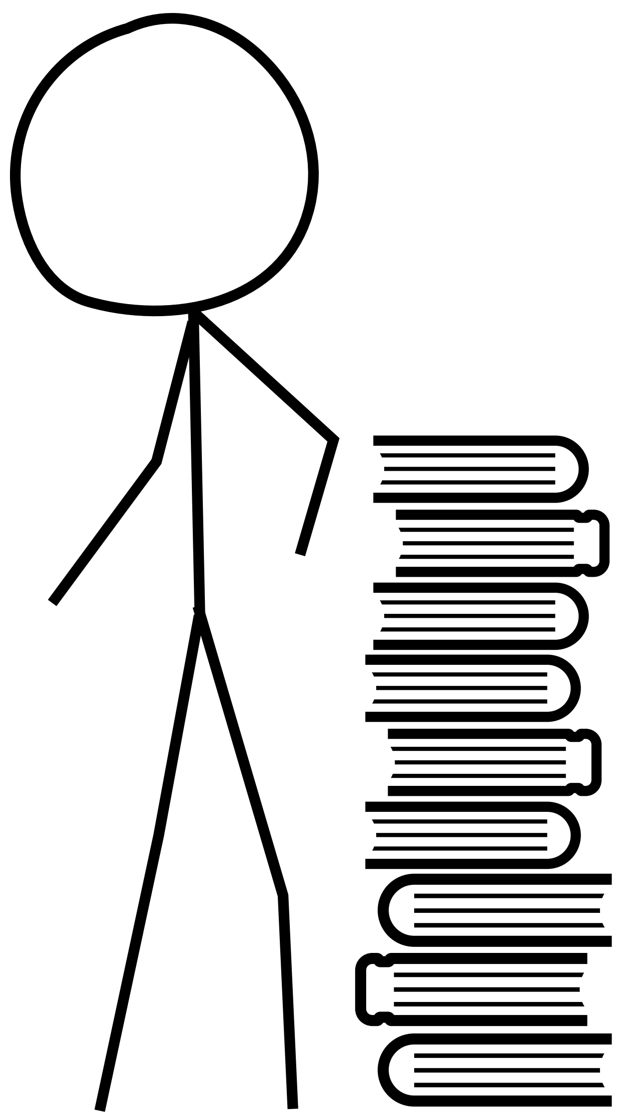
fetch([R, L], t_0) =



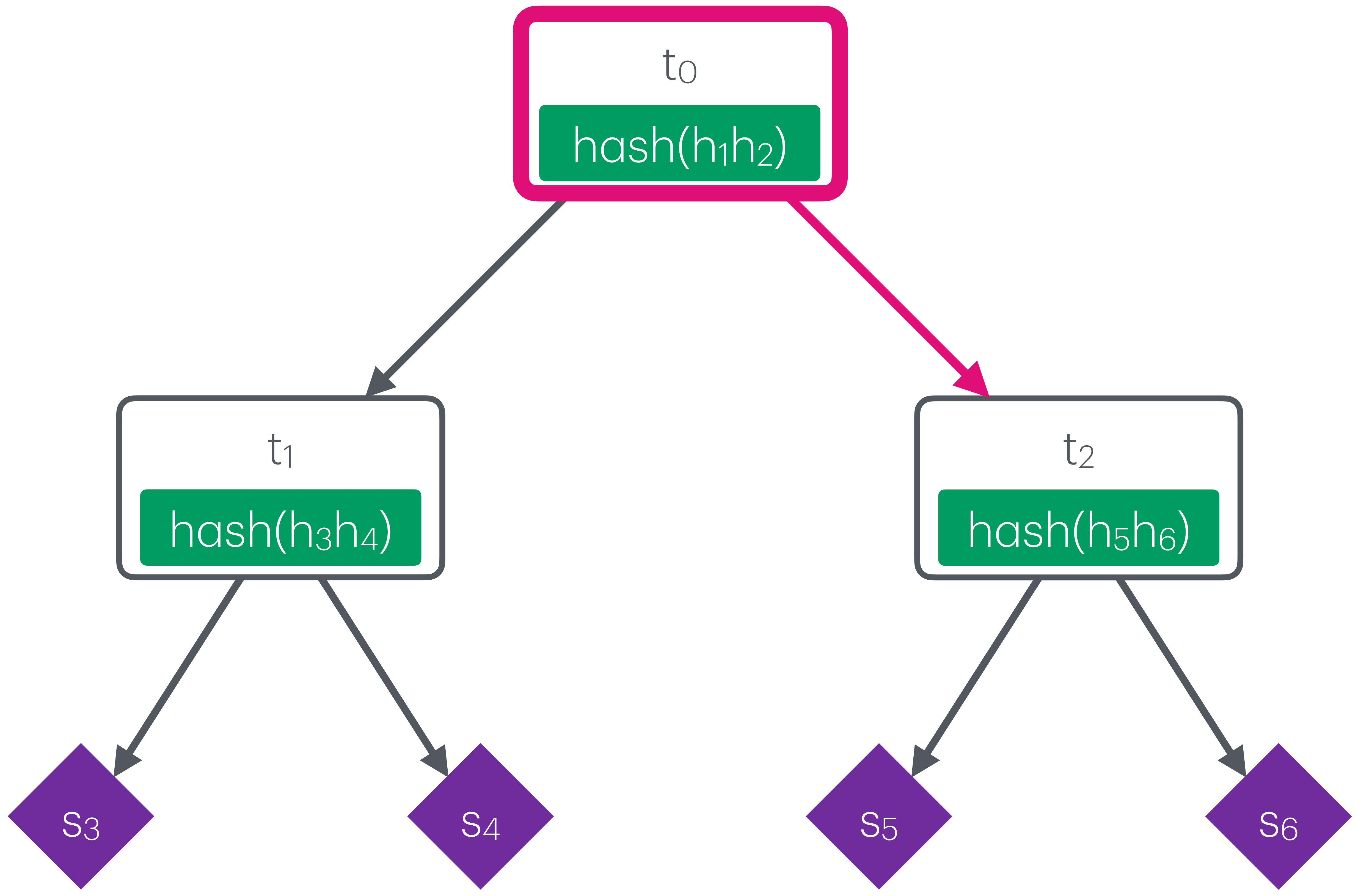
Example: Merkle Tree (Prover)



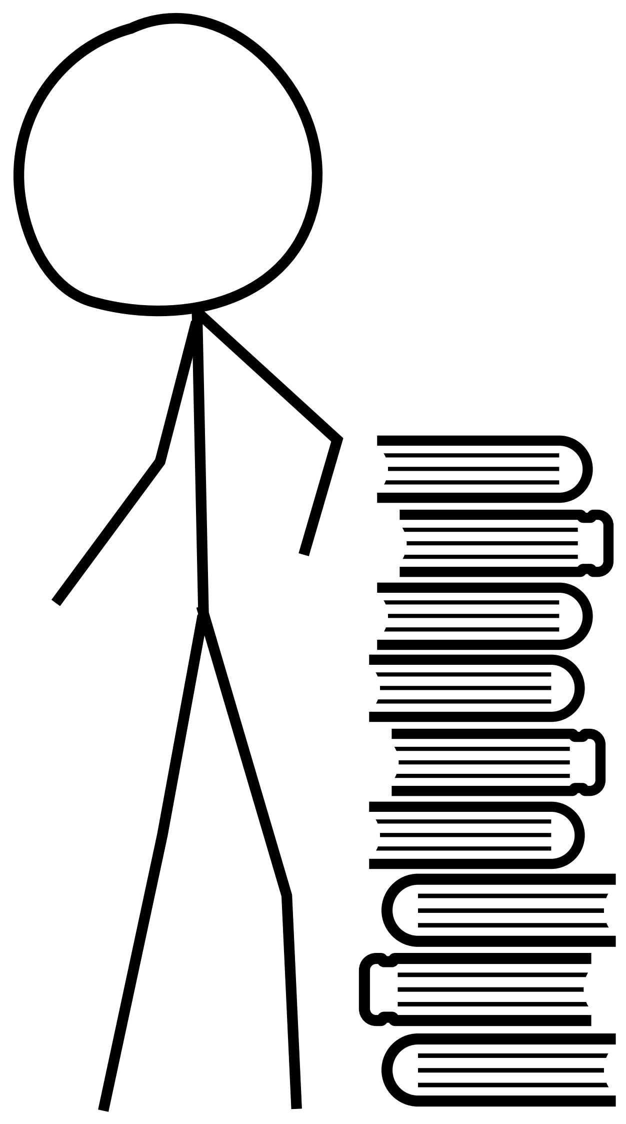
fetch([R, L], t_0) =



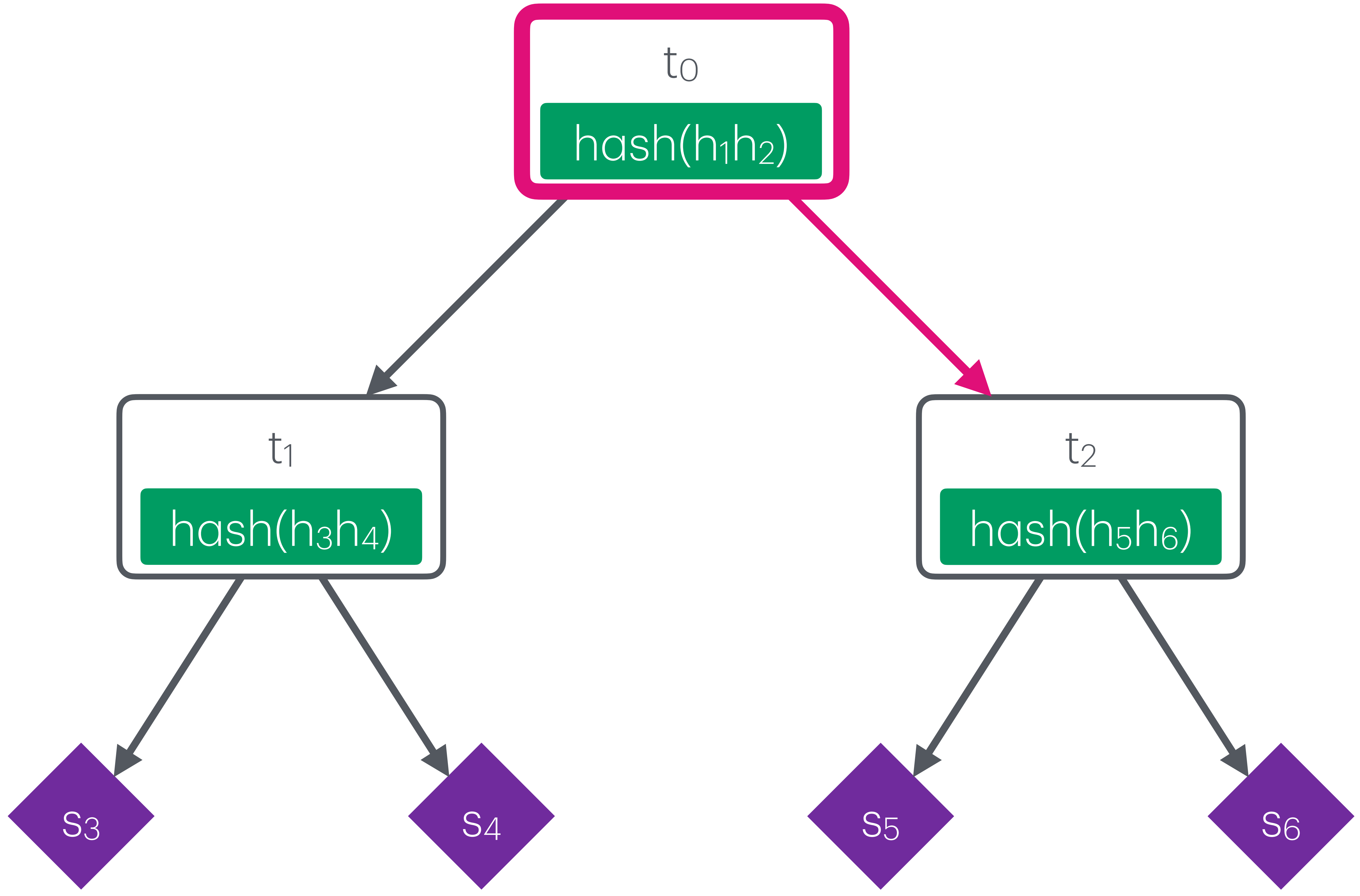
Example: Merkle Tree (Prover)



fetch([R, L], t_0) =

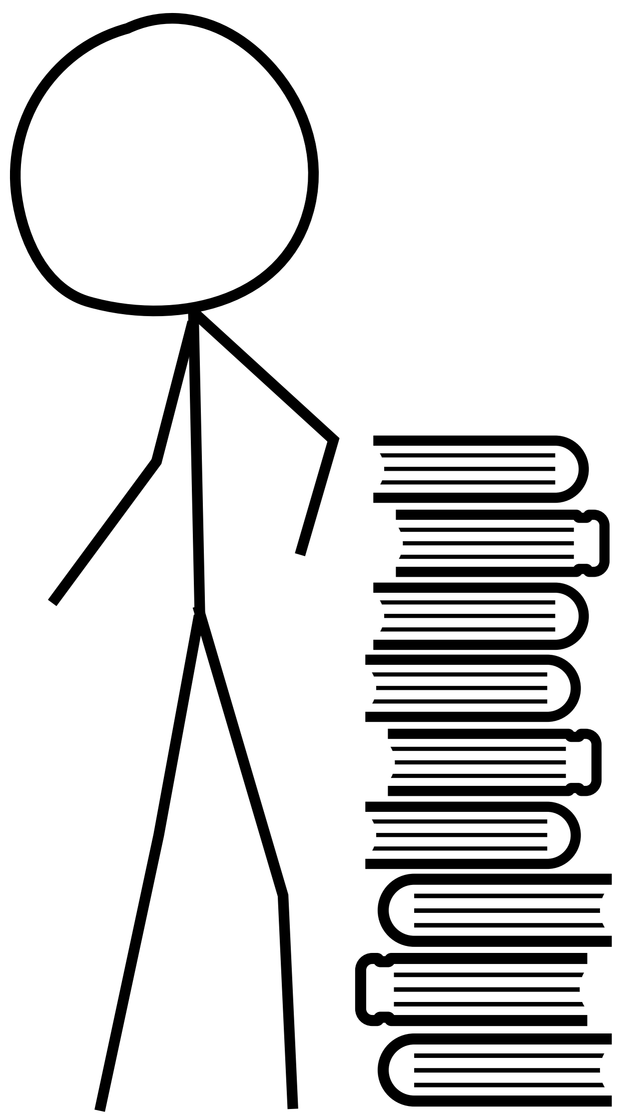


Example: Merkle Tree (Prover)

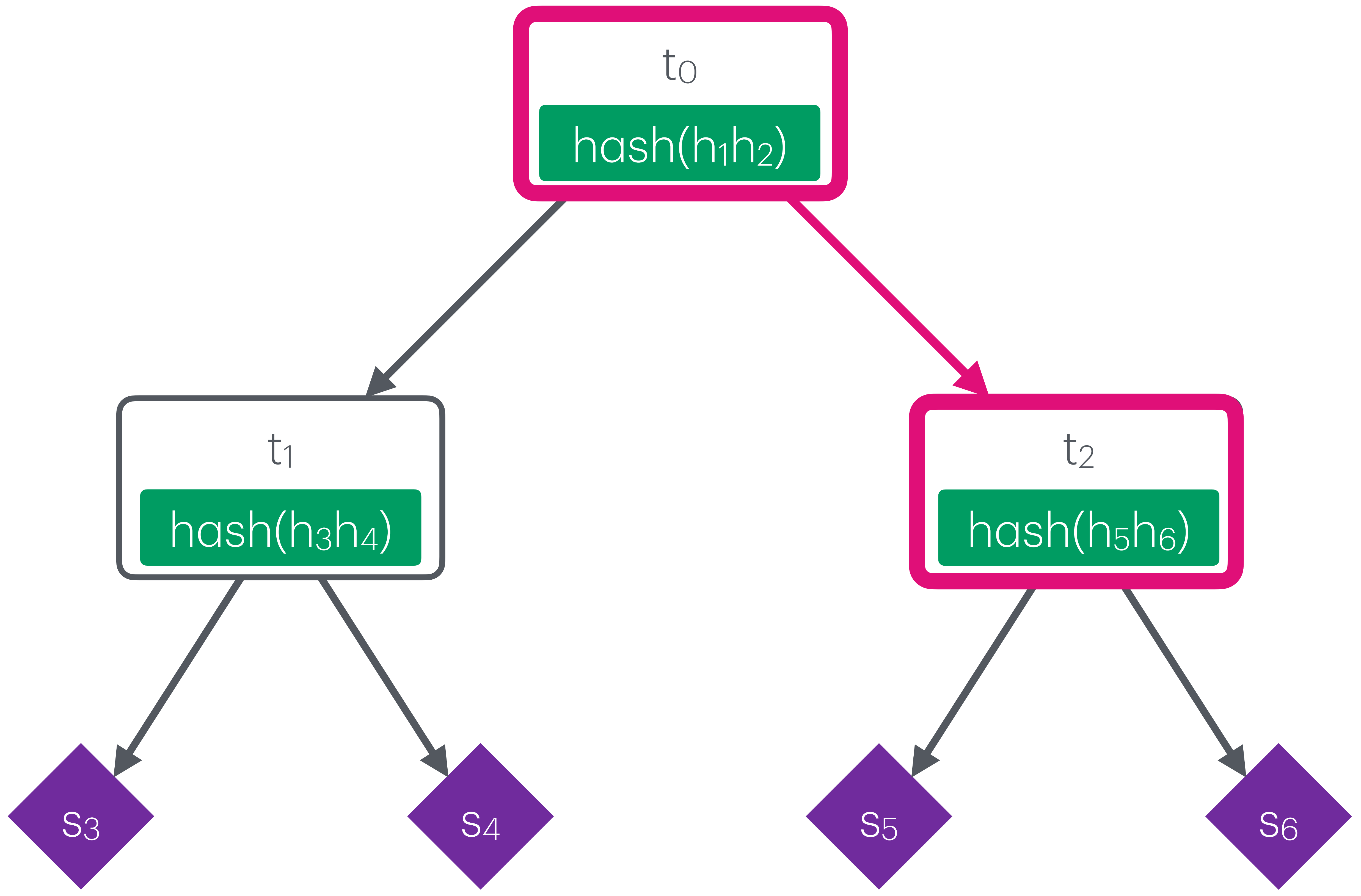


fetch([R, L], t_0) =

([h_1

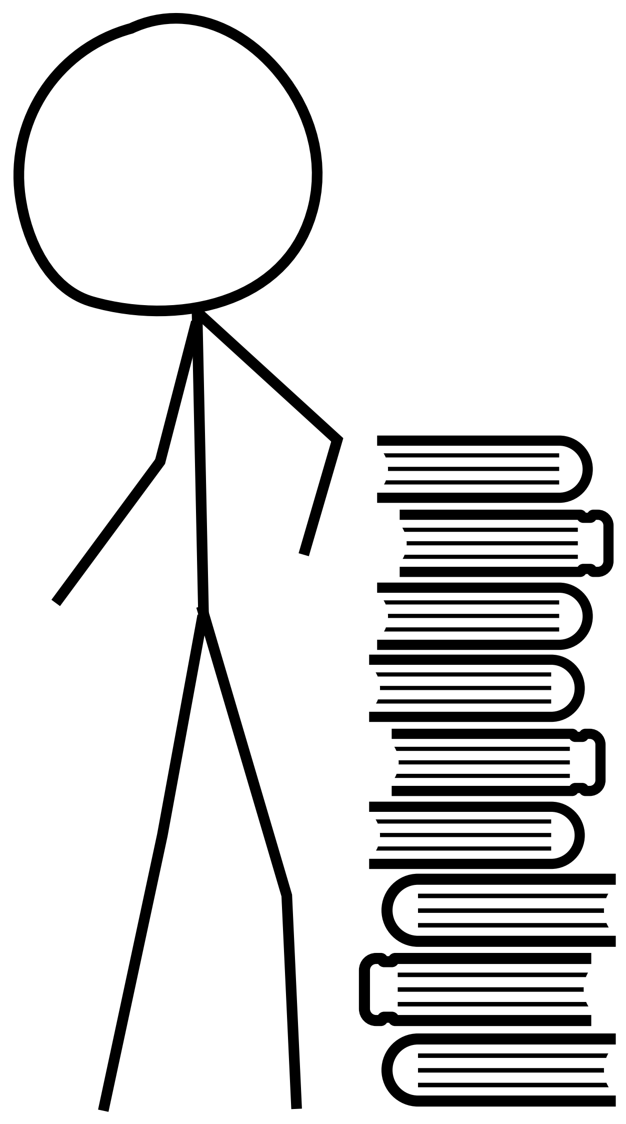


Example: Merkle Tree (Prover)

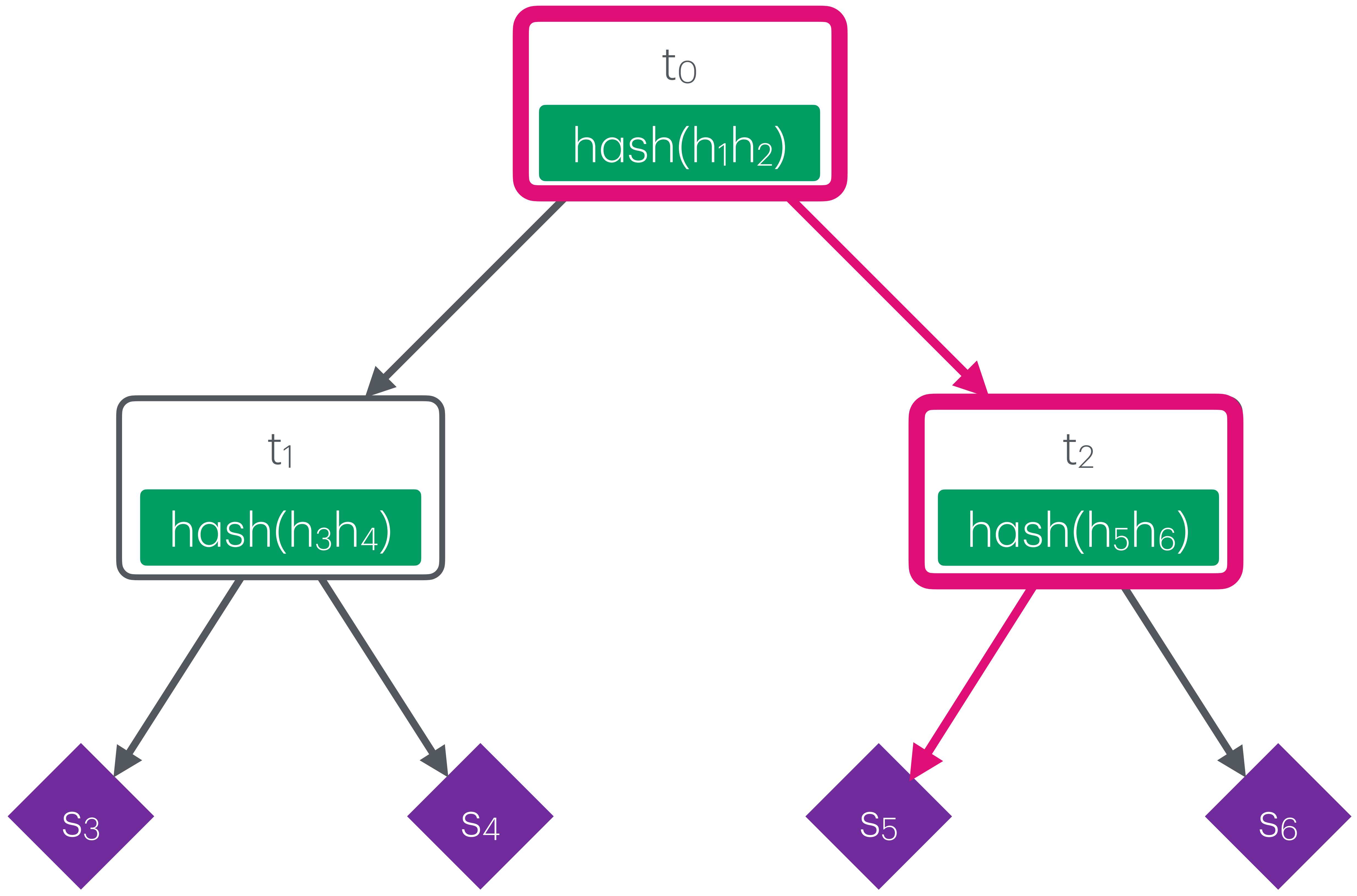


fetch([R, L], t_0) =

([h_1

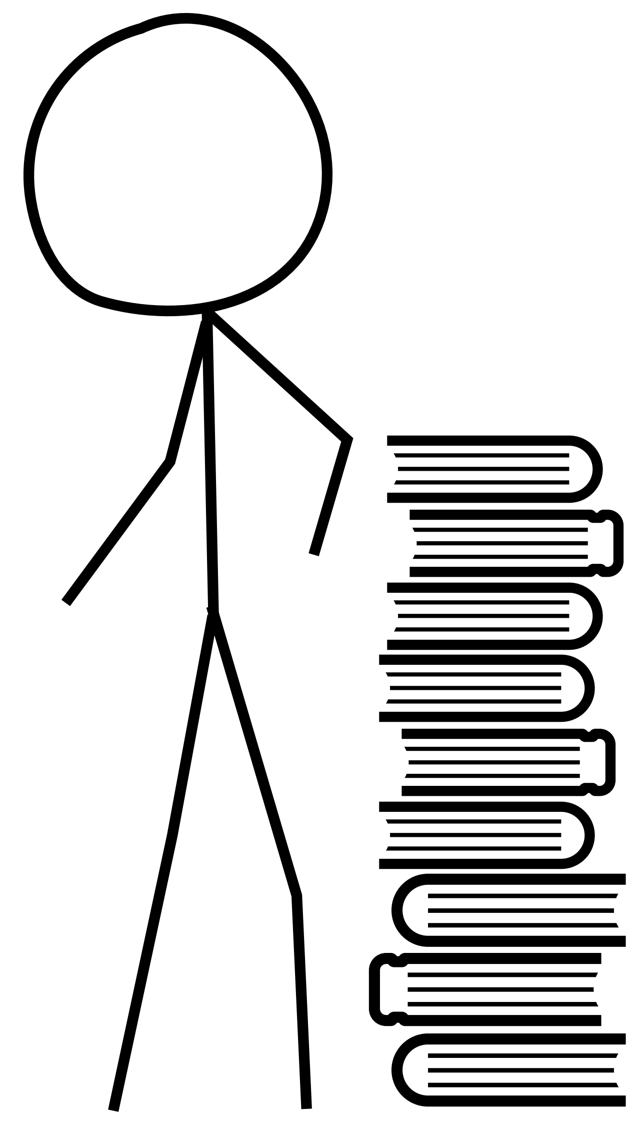


Example: Merkle Tree (Prover)

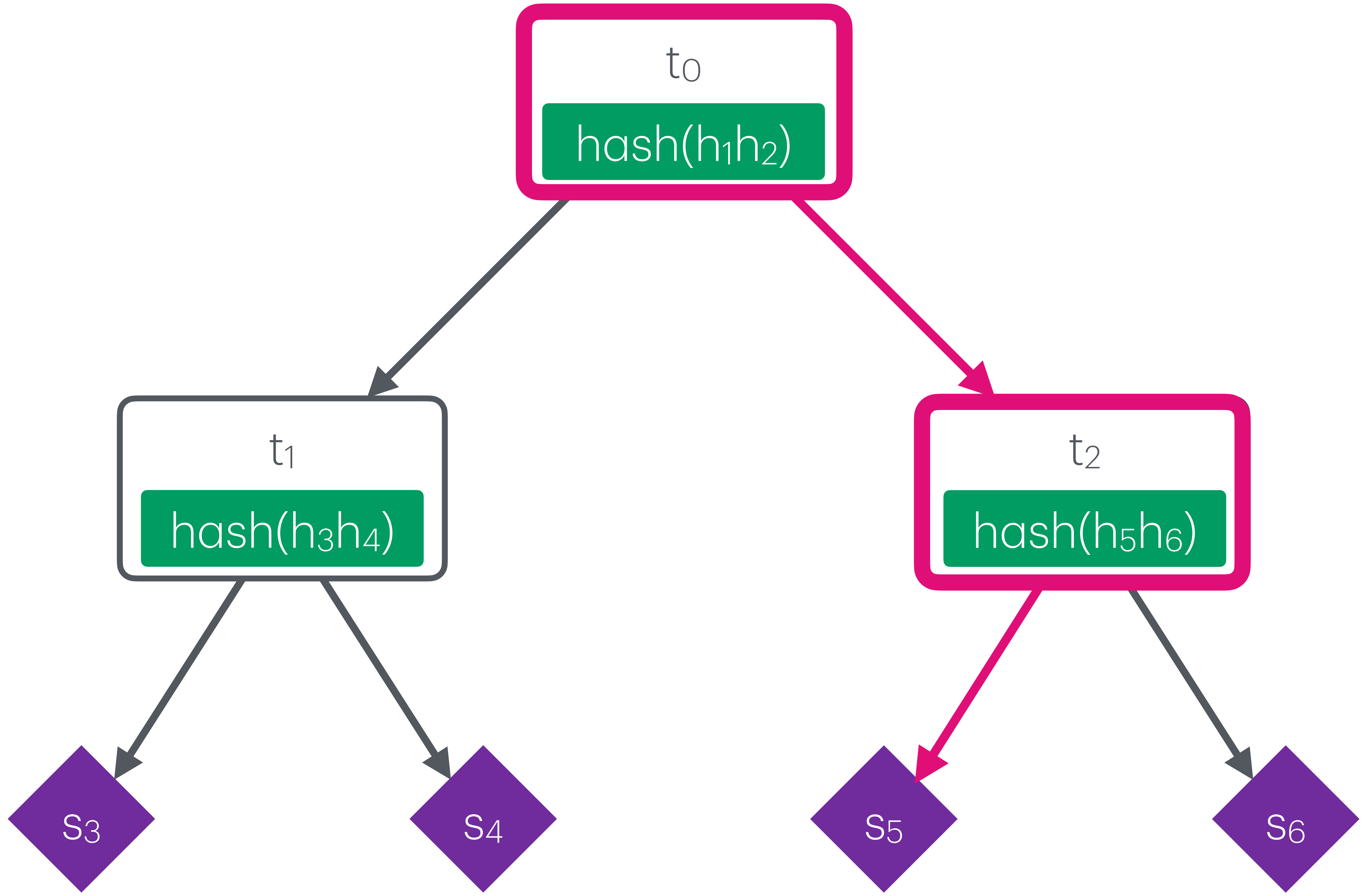


fetch([R, L], t_0) =

([h_1

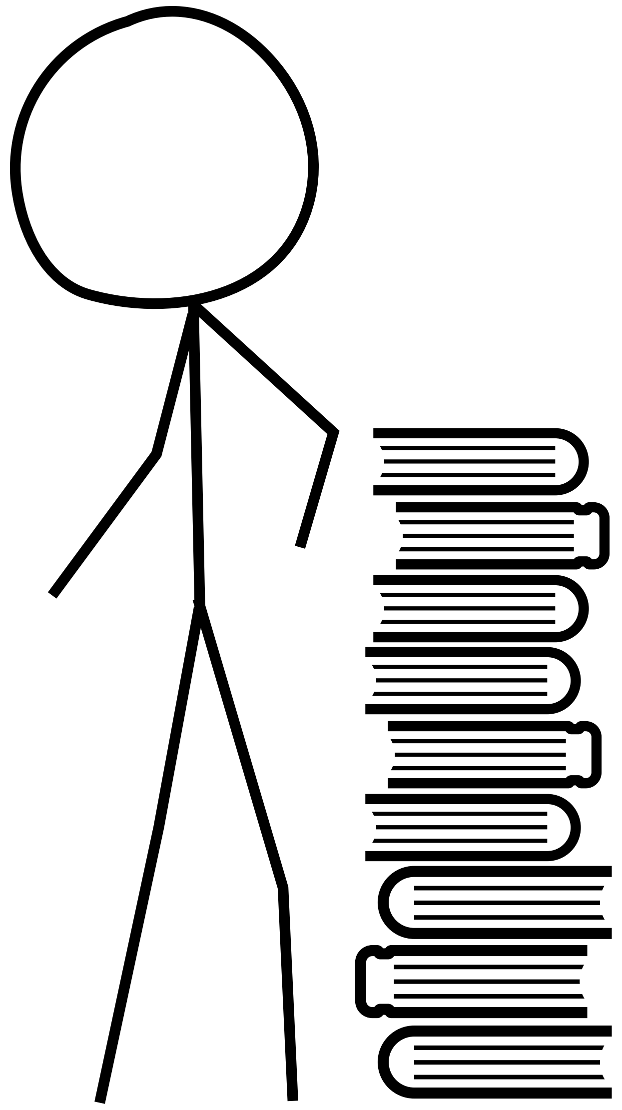


Example: Merkle Tree (Prover)

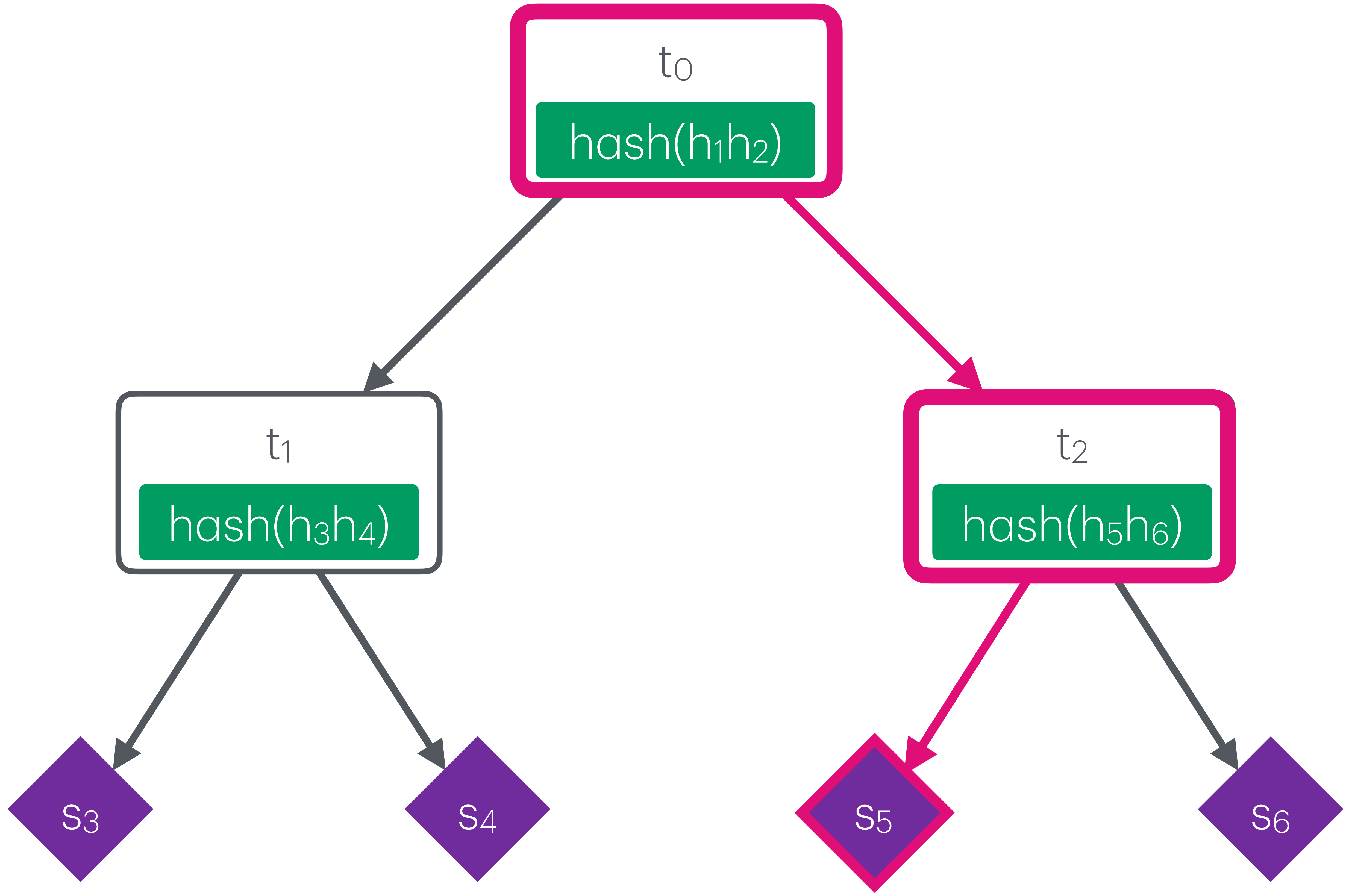


fetch([R, L], t_0) =

([h_1, h_6

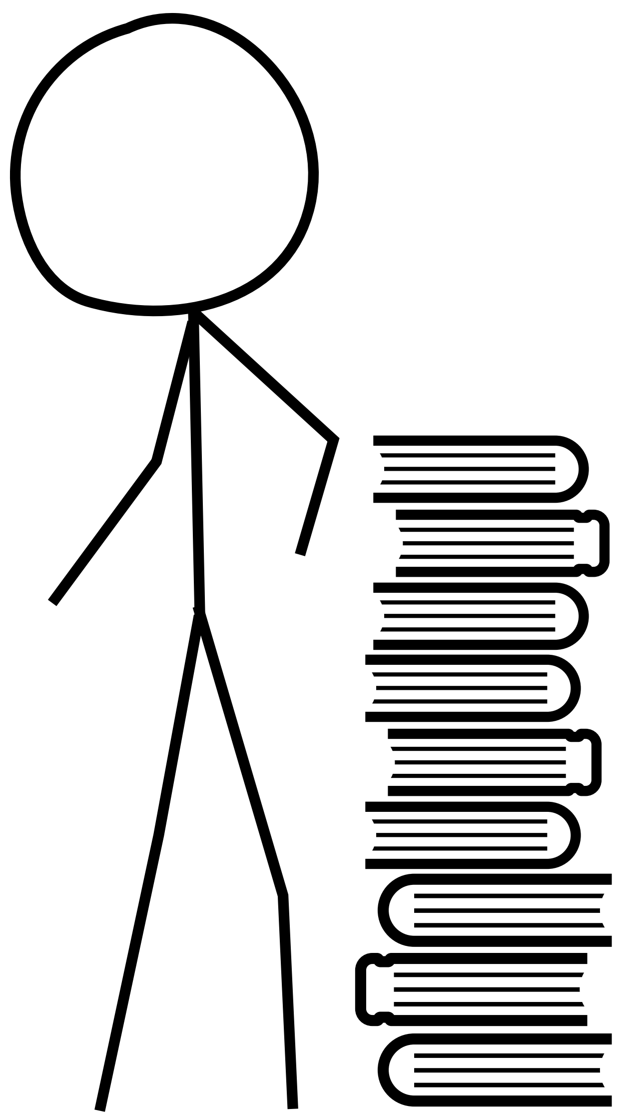


Example: Merkle Tree (Prover)

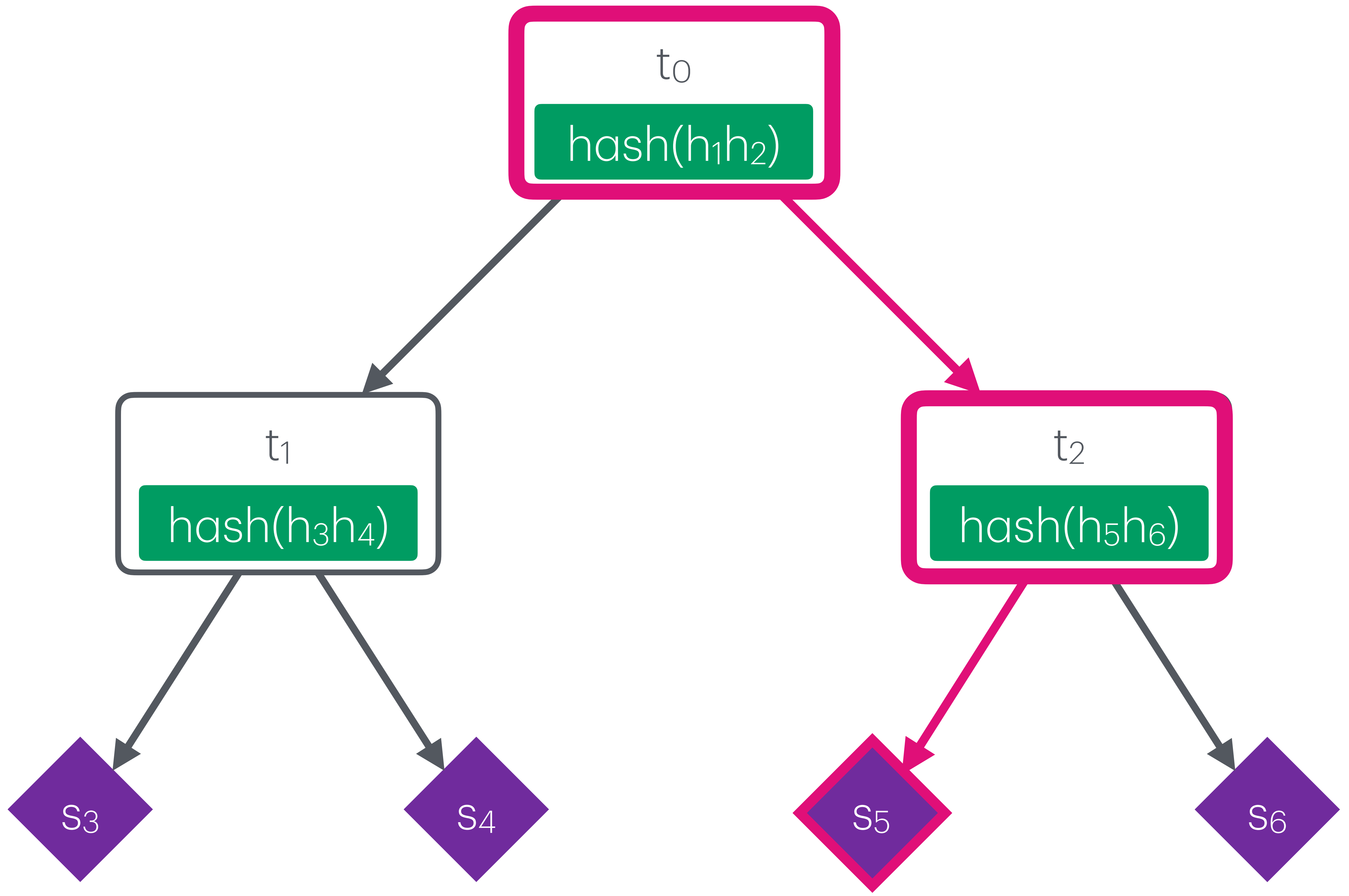


fetch([R, L], t_0) =

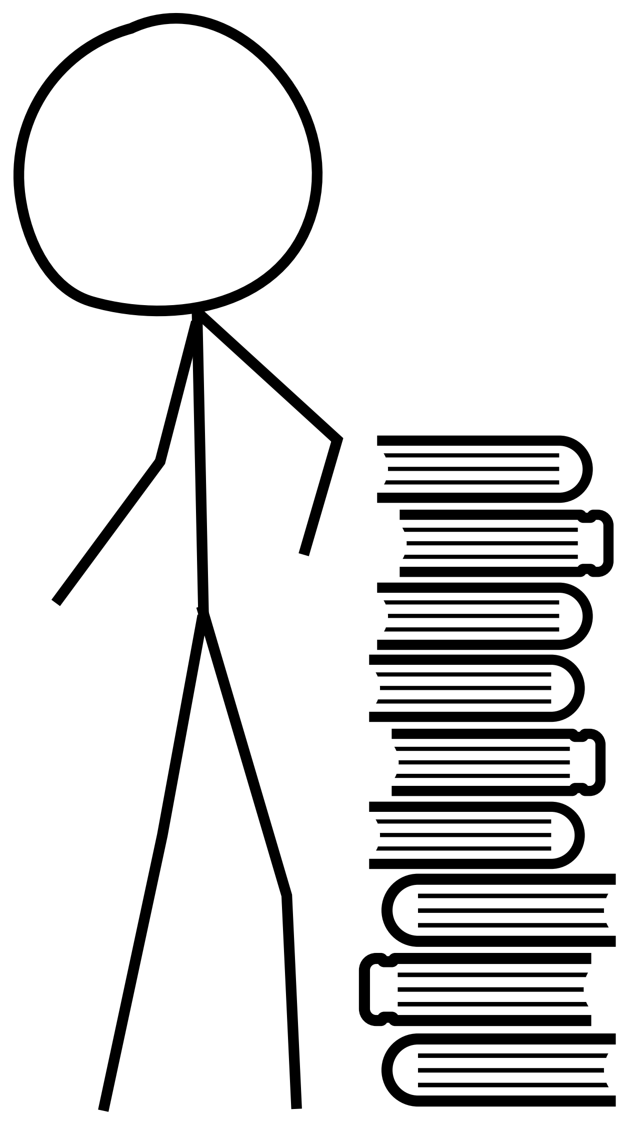
([h_1, h_6



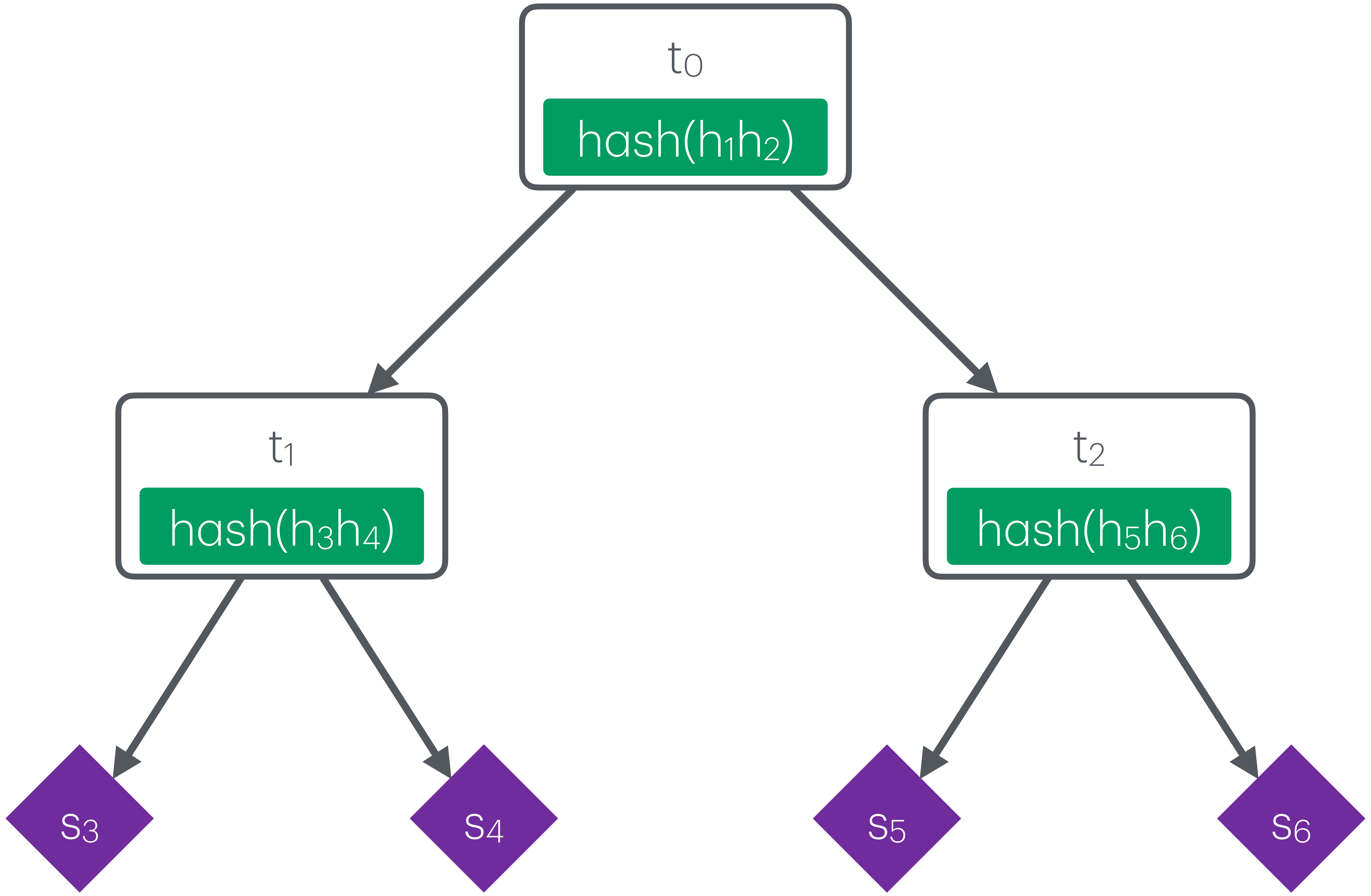
Example: Merkle Tree (Prover)



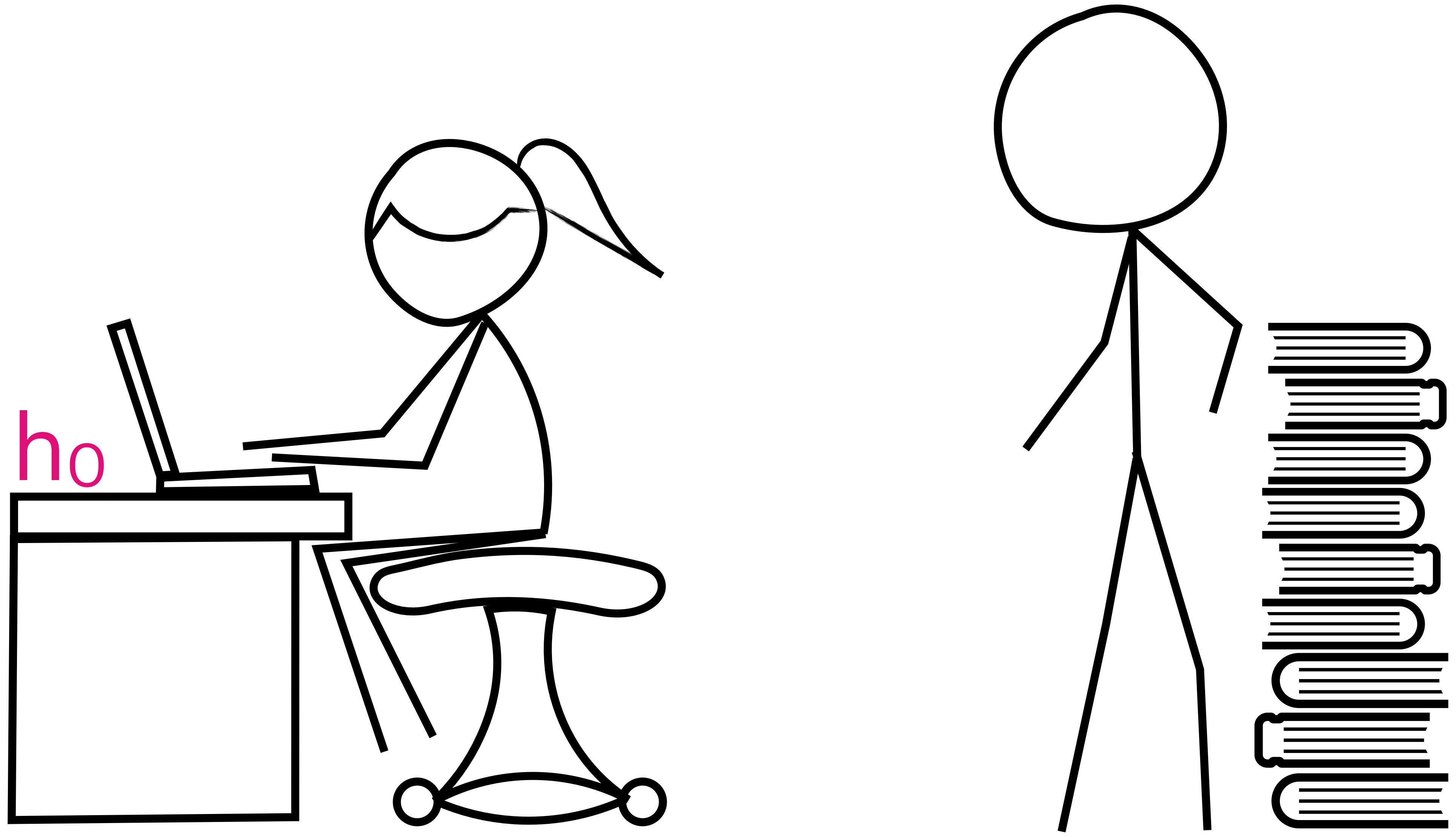
$$fetch([R, L], t_0) = ([h_1, h_6, s_5], s_5)$$



Example: Merkle Tree (Verifier)



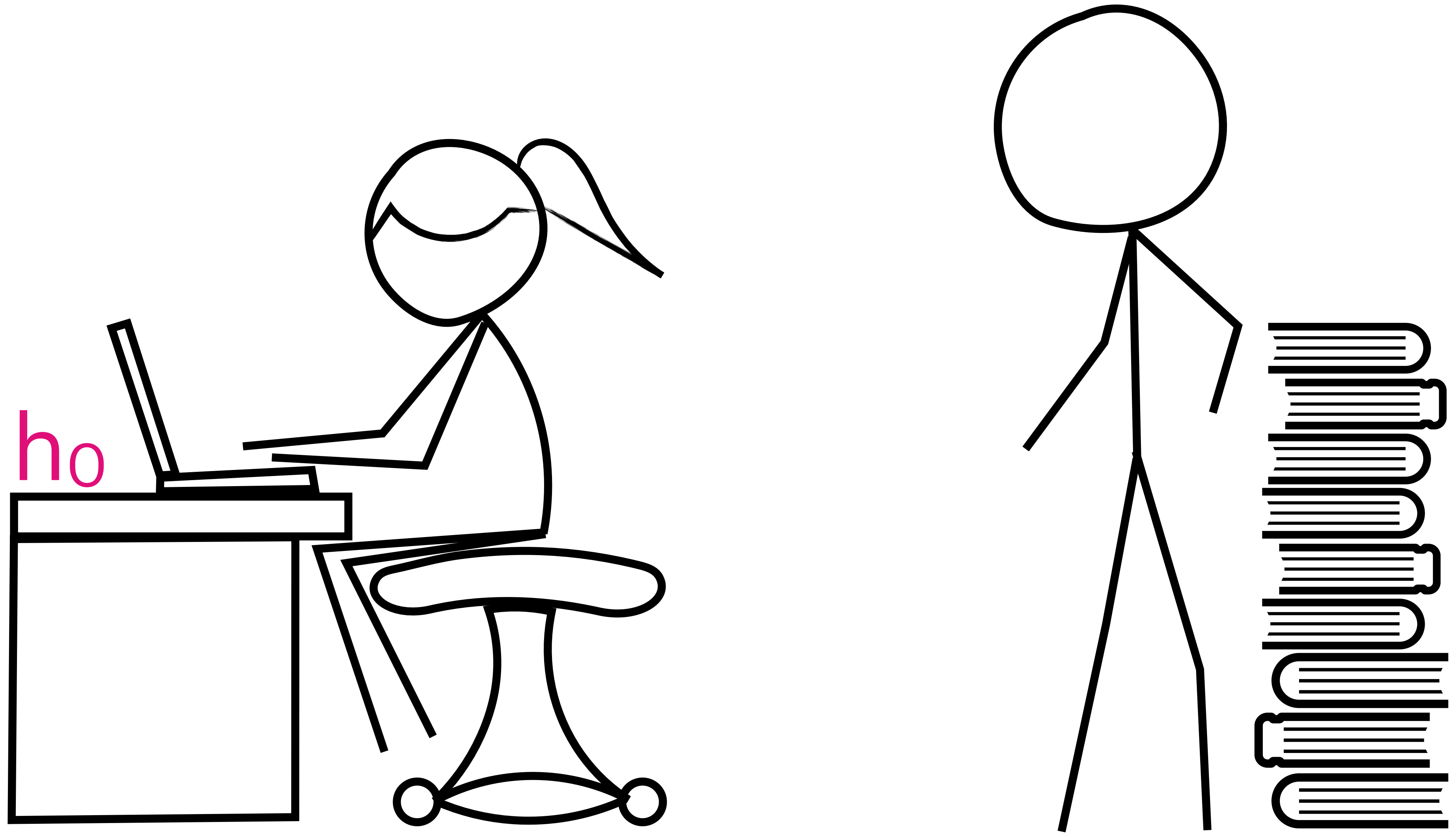
$$\text{fetch}([R, L], t_0) = ([h_1, h_6, s_5], s_5)$$



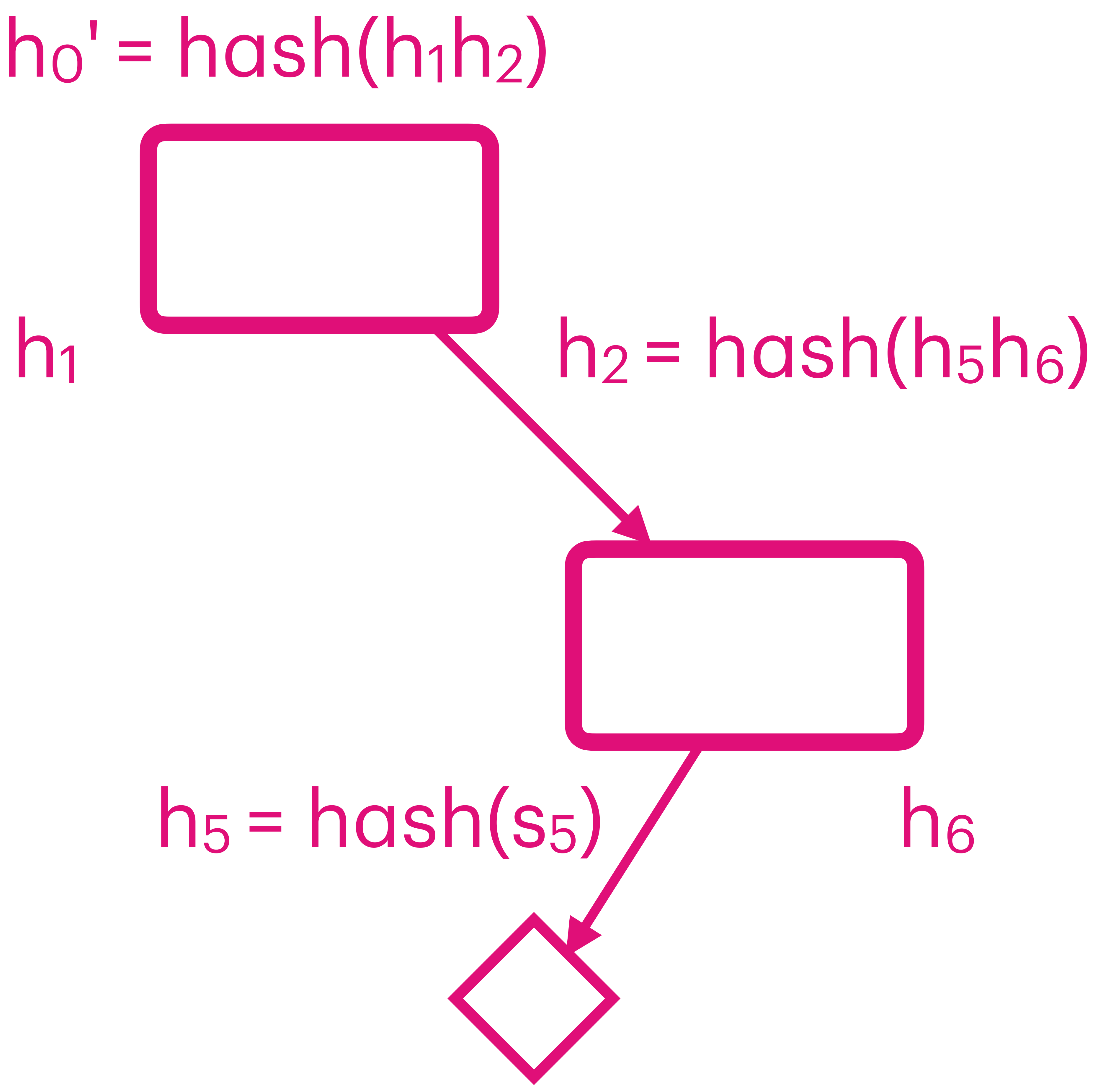
Example: Merkle Tree (Verifier)

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

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

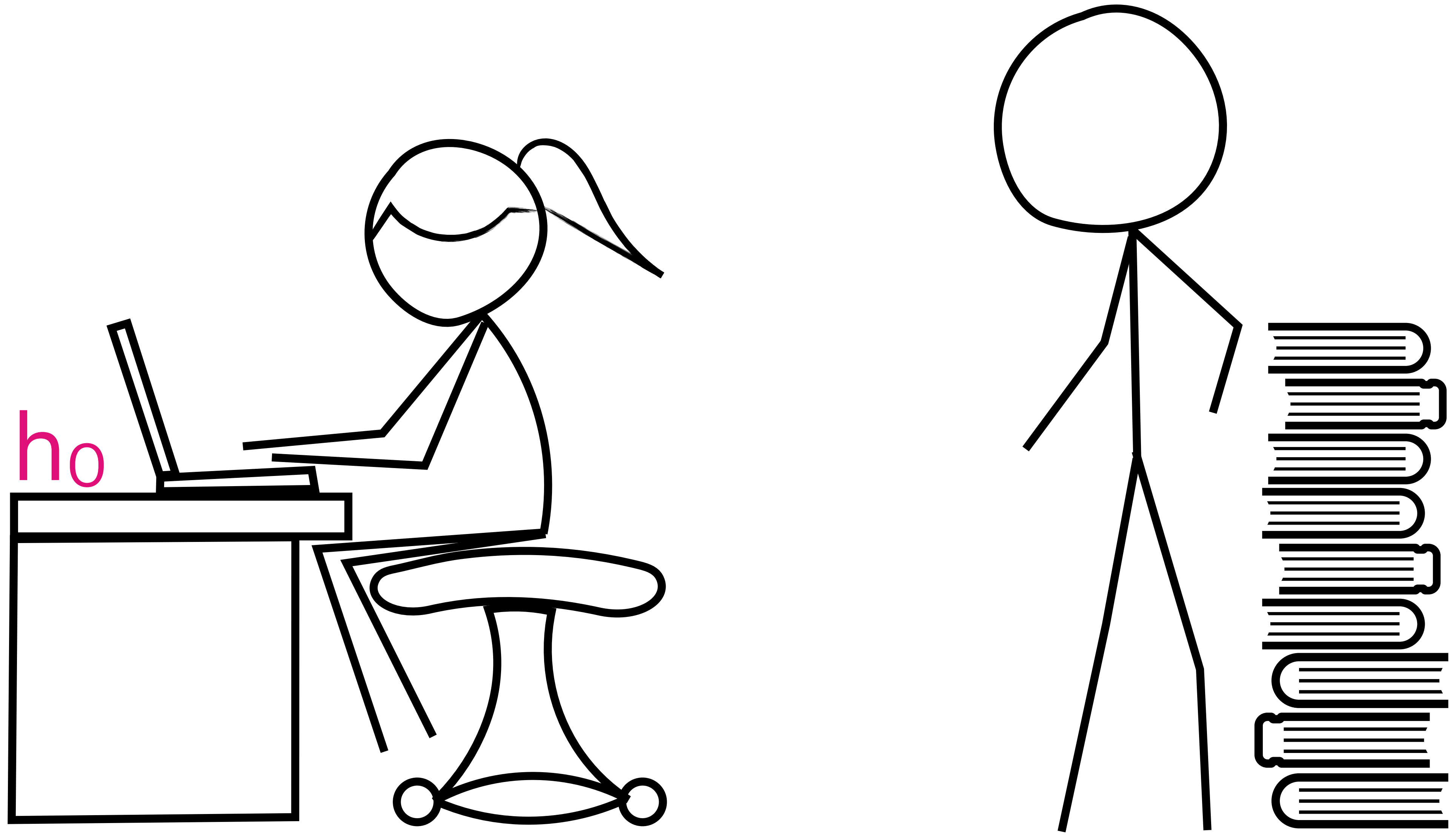


Example: Merkle Tree (Verifier)

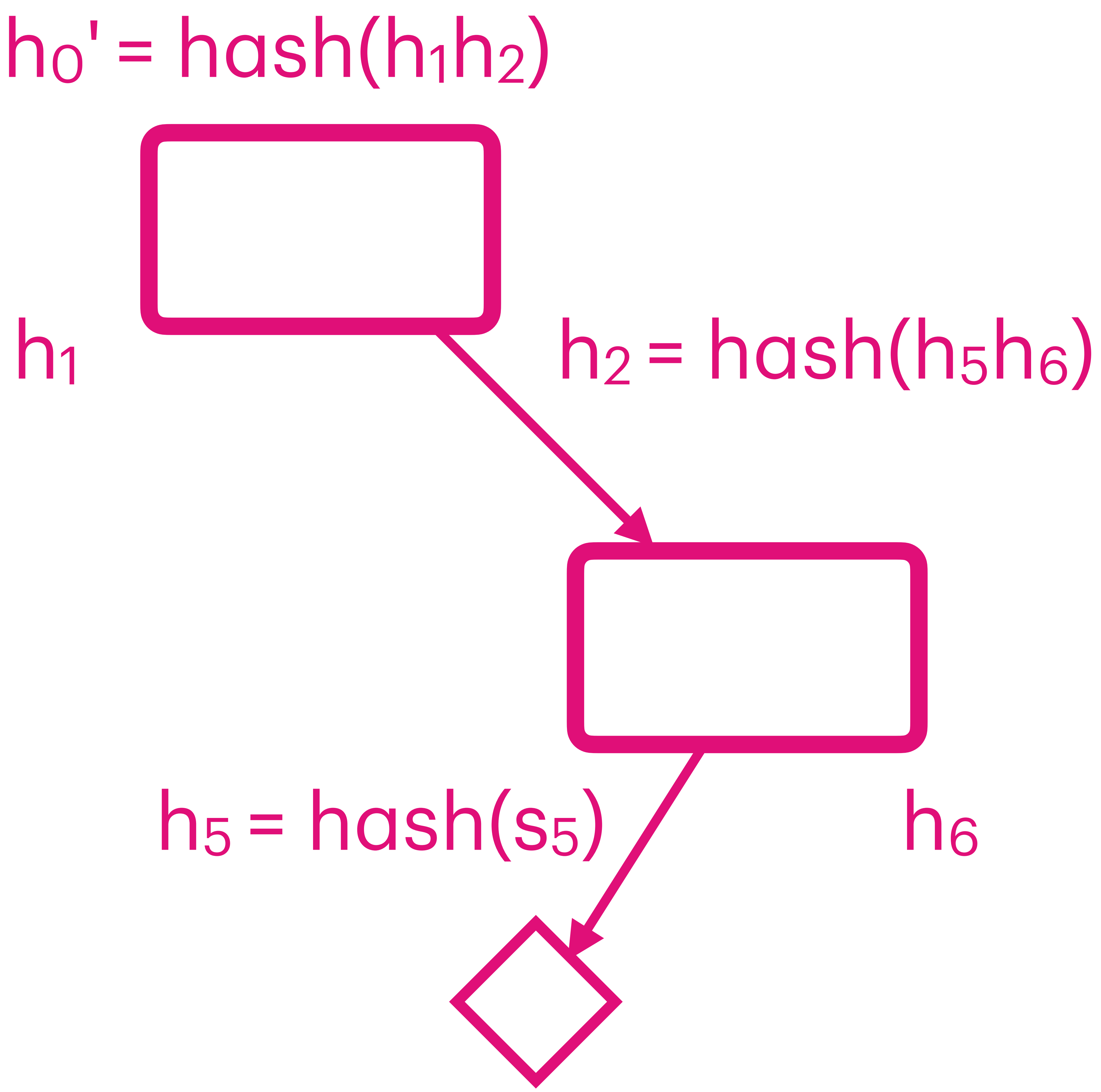


$fetch([R, L], t_0) =$

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

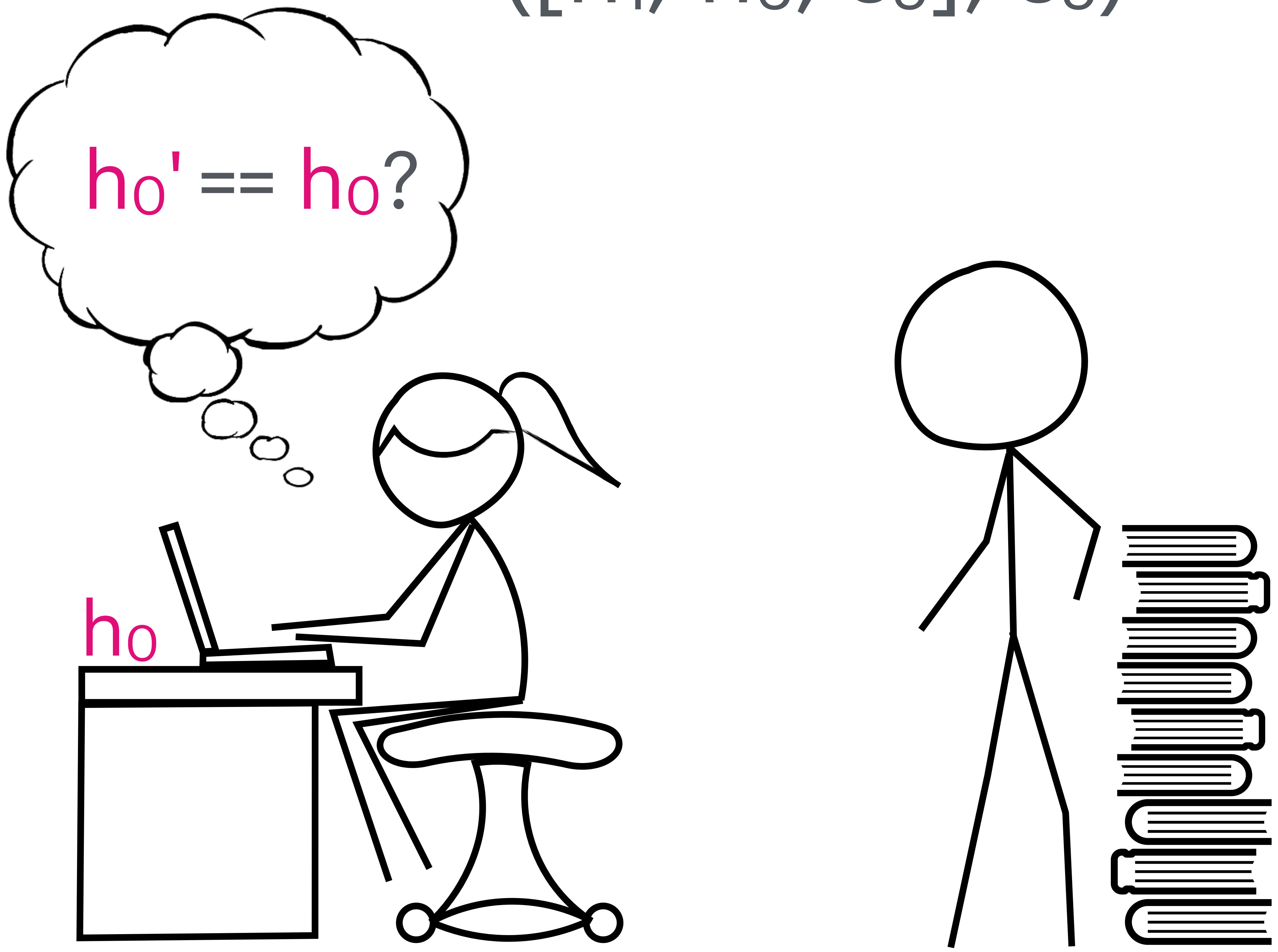


Example: Merkle Tree (Verifier)



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

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



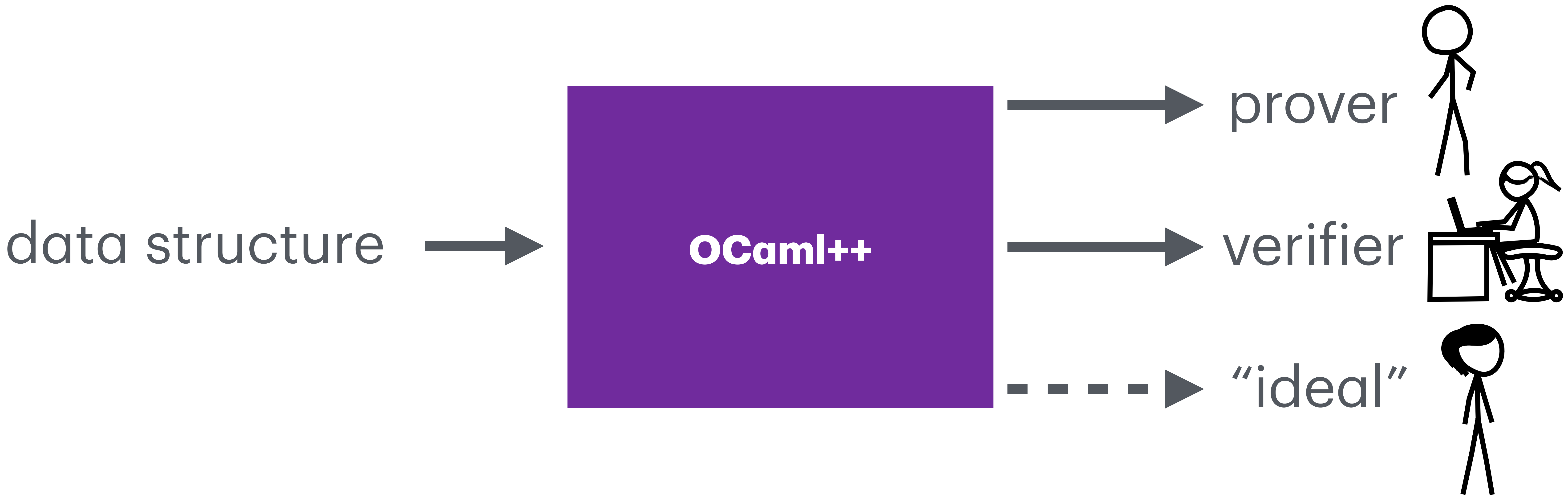
Use cases

- **Certificate transparency:** Google Chrome, Cloudflare, Let's Encrypt, Firefox, ...
- **Key transparency:** WhatsApp, Signal, ...
- **Binary transparency:** Google 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.



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.

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 do not 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

About | Blog | Publications

BOB ATKEY

Authenticated Data Structures, as a Library, for Free!

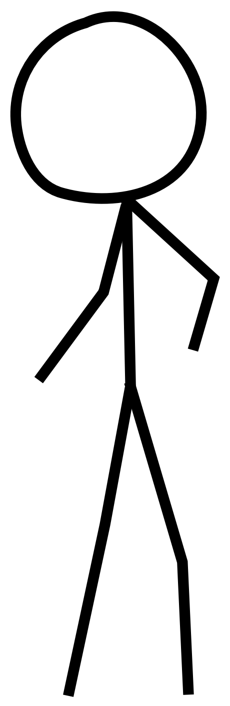
Let's assume that you're querying to some database stored in the cloud (i.e., on someone else's computer).

Published: Tuesday 12th April 2016

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

  (* ... *)

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

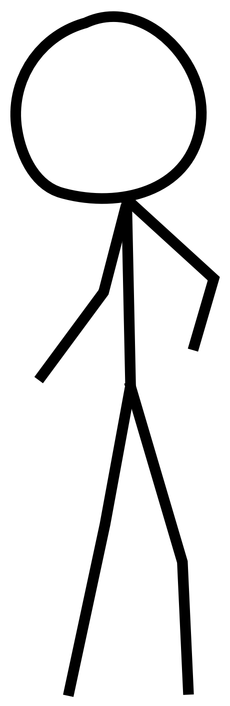
```
module Merkle = functor (A : AUTHENTIKIT) -> struct
  open A

  (* ... *)

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

```
module Prover : AUTHENTIKIT
```



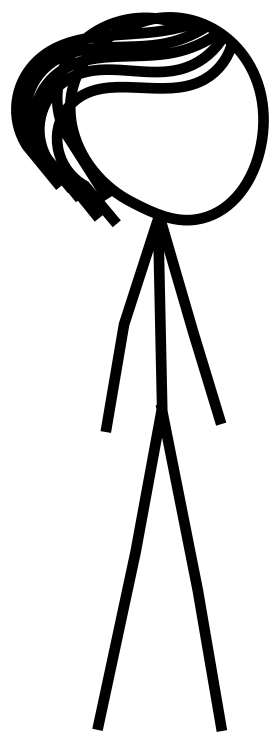
```
module Merkle = functor (A : AUTHENTIKIT) -> struct
  open A

  (* ... *)

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

```
module Prover : AUTHENTIKIT
```



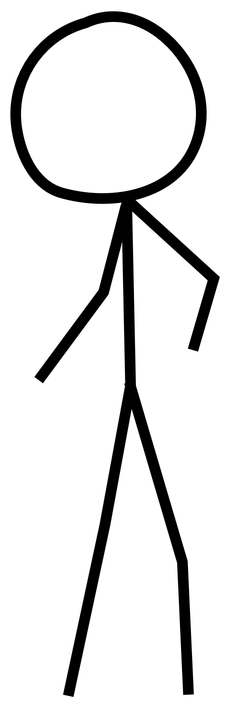
```
module Merkle = functor (A : AUTHENTIKIT) -> struct
  open A

  (* ... *)

  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

- A proof of **security and correctness** of the Authentikit construction.
- We address the remaining two limitations:
  - We verify several **optimizations** of Authentikit.
  - We show how to **safely link** manually verified code with code automatically generated by Authentikit through semantic typing.
- Full mechanization in the Rocq prover using the Iris framework.

```
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
```

```
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]
  type tree  = 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]
  type tree  = tree' auth

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


  (* ... *)




  (* ... *)




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]
  type tree  = tree' auth

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

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




  (* ... *)




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]
  type tree  = tree' auth

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

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

  let rec fetch (p : path) (t : tree) : 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
```

Takeaway

Takeaway

- In the end, it is not so difficult to prove that **one particular client** has the security and correctness property.

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!

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 the module system.
  In fact, we prove security and correctness as so-called “free” theorems.

Our approach

Our approach

To show, e.g., security, we

Our approach

To show, e.g., security, we

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

$$\{P\}\, e_1 \sim e_2\, \{Q\}$$

Our approach

To show, e.g., security, we

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

$$\{P\}\, e_1 \sim e_2\, \{Q\}$$

2. Show that syntactically well-typed programs satisfy a CSFL judgment.

$$\Gamma, A : \mathsf{Authentikit} \vdash e : \tau \qquad \Rightarrow \qquad \{\ldots\}\, e \sim e\, \{\ldots\}$$

Our approach

To show, e.g., security, we

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

$$\{P\}\, e_1 \sim e_2 \,\{Q\}$$

2. Show that syntactically well-typed programs satisfy a CSFL judgment.

$$\Gamma, A : \mathsf{Authentikit} \vdash e : \tau \qquad \Rightarrow \qquad \{\ldots\}\, e \sim e \,\{\ldots\}$$

3. **Verify implementations** of Authentikit in CSFL.

$$\{\ldots\}\, \mathsf{Verifier} \sim \mathsf{Ideal} \,\{[\![\mathsf{Authentikit}]\!]\}$$

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.,

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

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

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

then for all proofs p, if

Theorem (Security)

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

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

then for all proofs p, if

$$e \text{ instantiated with } \mathsf{Verifier} \text{ accepts } p \text{ and returns } v$$

Theorem (Security)

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

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

then for all proofs p, if

$$e \text{ instantiated with } \mathsf{Verifier} \text{ accepts } p \text{ and returns } v$$

then

Theorem (Security)

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

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

then for all proofs p, if

$$e \text{ instantiated with } \mathsf{Verifier} \text{ accepts } p \text{ and returns } v$$

then

- e instantiated with Ideal returns v or

Theorem (Security)

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

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

then for all proofs p, if

$$e \text{ instantiated with } \mathsf{Verifier} \text{ accepts } p \text{ and returns } v$$

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.,

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

then if

$$e \text{ instantiated with } \mathsf{Prover} \text{ produces a proof } p \text{ and returns } v$$

then

- e instantiated with $\mathsf{Verifier}$ accepts p and returns v and
- e instantiated with Ideal returns v.

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.

Summary

- **Authentikit** is a library for implementing secure and correct ADSs generically.
- **A proof of security and correctness** of the Authentikit construction.
  - We verify several **optimizations**.
  - We show how to **safely link** manually verified code with code automatically generated using Authentikit.
- Full mechanization in the Rocq prover using the Iris framework.

https://simongregersen.com/papers/2025-authentikit.pdf


That’s it, folks!

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

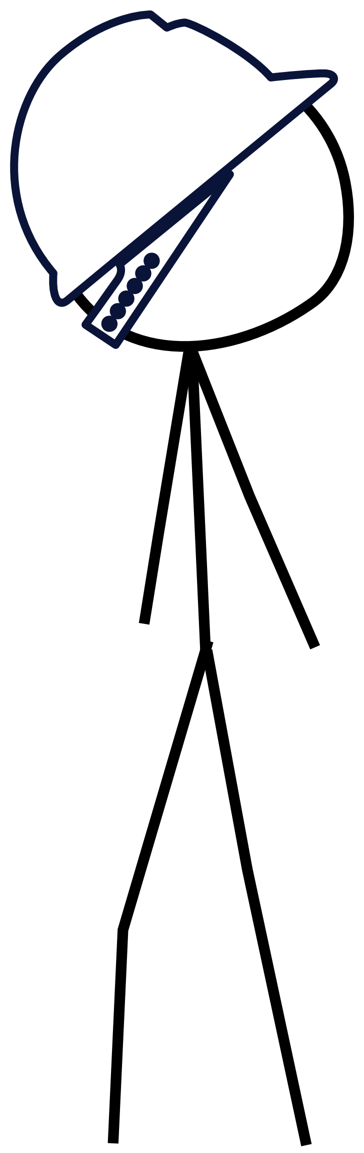
Miller et al.'s approach

OCaml is extended with three new primitives:

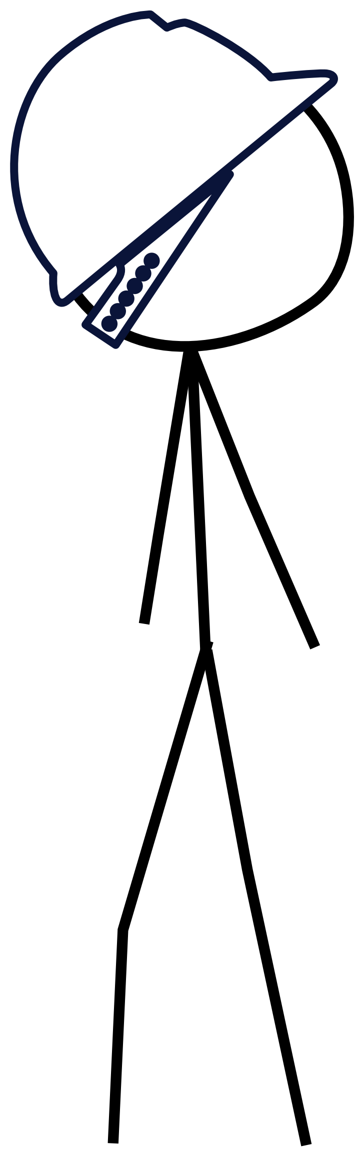
- authenticated types $\bullet\,\tau$
- auth : 'a → • 'a
- unauth : • 'a → 'a

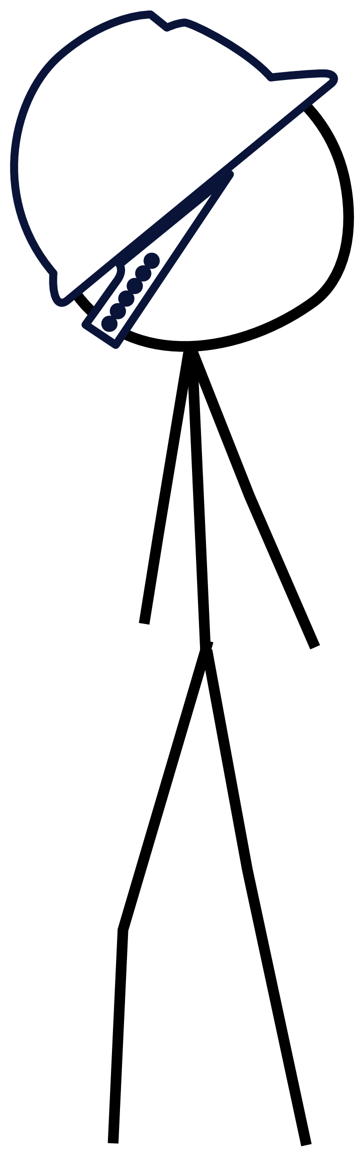
```
type tree = Tip of string | Bin of •tree × •tree
type bit = L | R
let rec fetch (idx:bit list) (t:•tree) : string =
  match idx, unauth t with
  | [], Tip a → a
  | L :: idx, Bin(l,_) → fetch idx l
  | R :: idx, Bin(_,r) → fetch idx r
```

```
type proof = string list

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







  (* ... *)







  (* ... *)

end
```



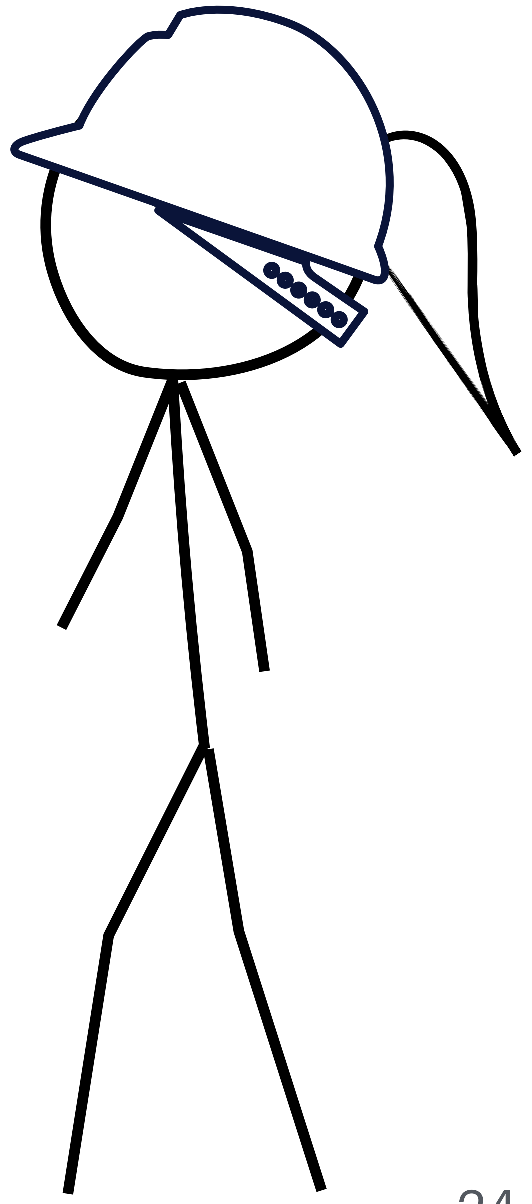
```
type proof = string list

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
```



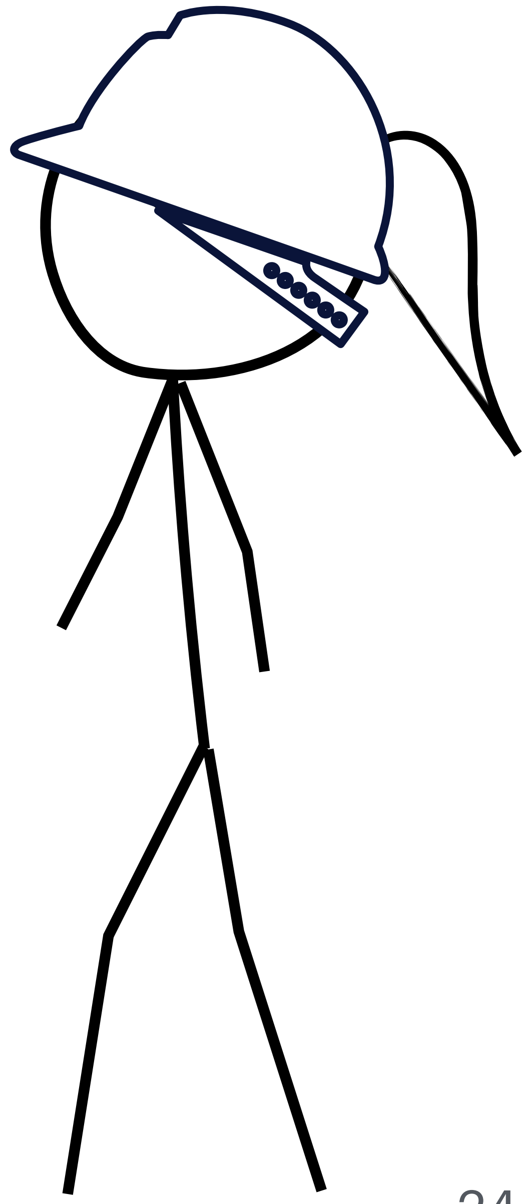


```
type proof = string list

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
```

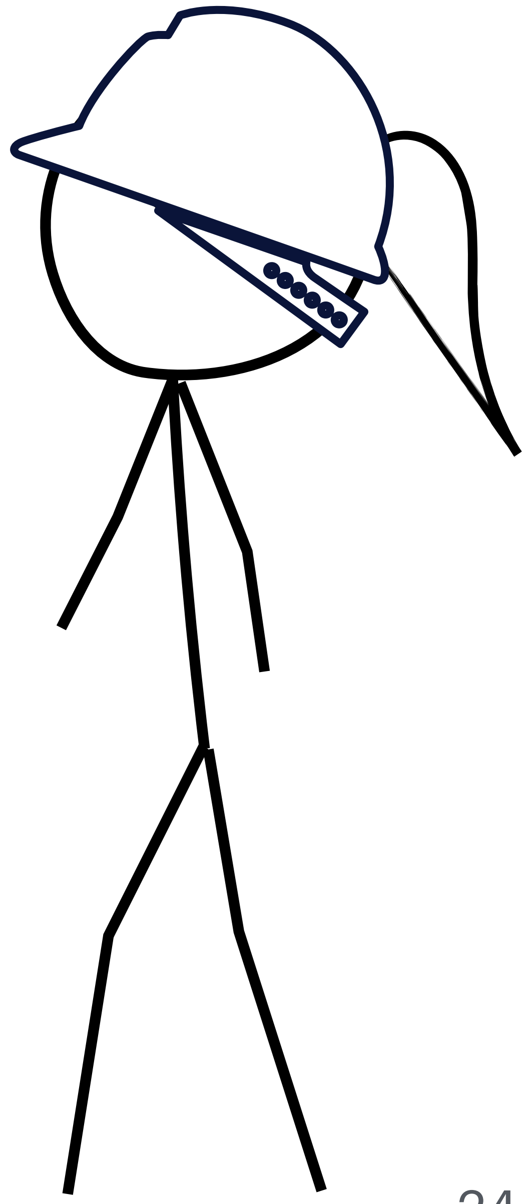
```
module Verifier : AUTHENTIKIT =
  type 'a auth = string
  type 'a auth_computation =
    proof -> [`Ok of proof * 'a | `ProofFailure]







  (* ... *)










  (* ... *)




end
```



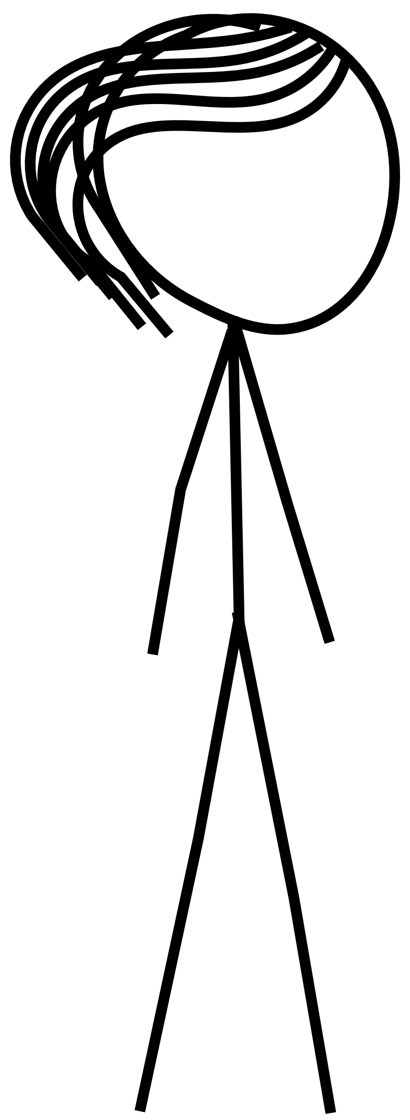
```
module Verifier : AUTHENTIKIT =
  type 'a auth = string
  type 'a auth_computation =
    proof -> [`Ok of proof * 'a | `ProofFailure]

  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
```



```ocaml
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
```



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

```
module Prover : AUTHENTIKIT
```

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
```

```
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 Prover : AUTHENTIKIT
```

abstract type constructors

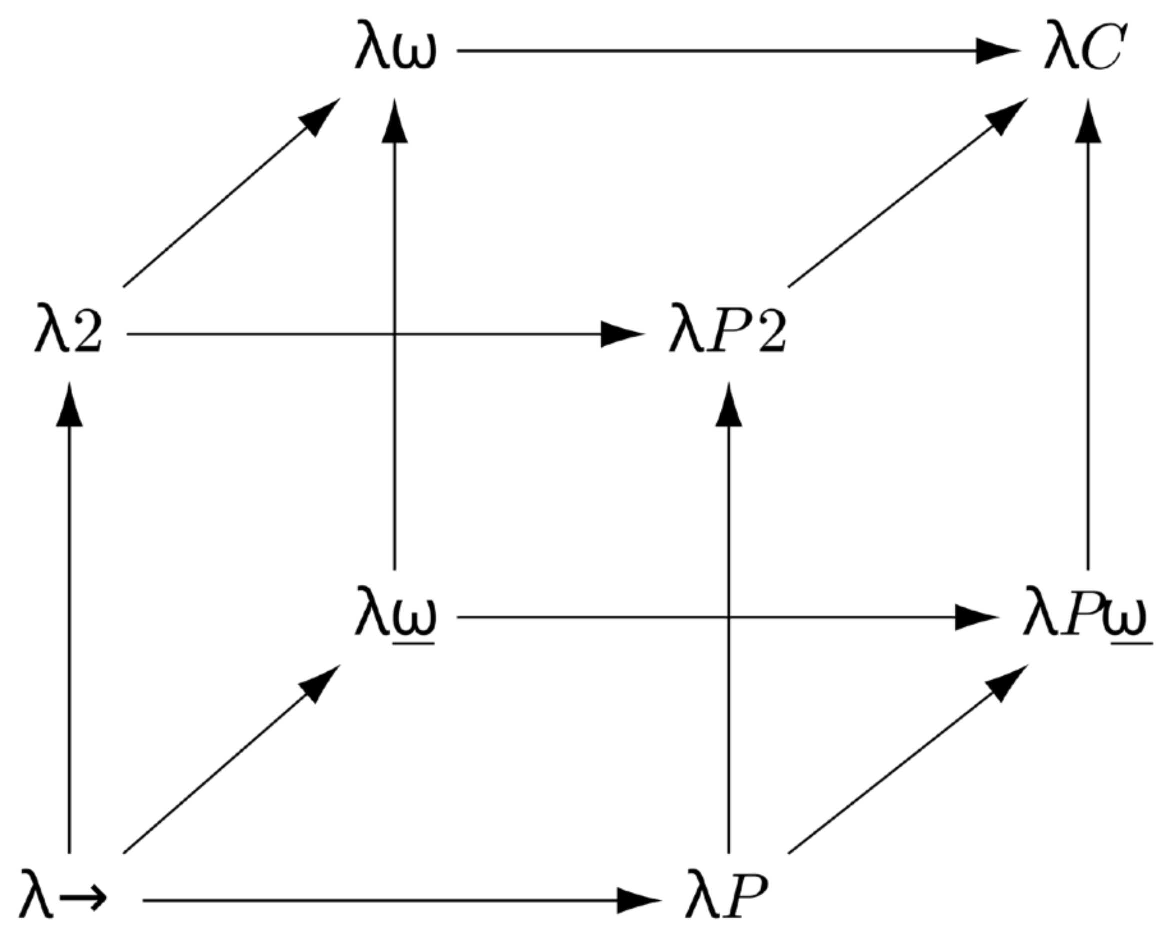
recursive types

(higher-order) functions

state

polymorphism

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,

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

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

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

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

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

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

$\kappa ::= \star \mid \kappa \Rightarrow \kappa$ (kinds)

$\tau ::= \alpha \mid \lambda\alpha : \kappa\,.\,\tau \mid \tau\ \tau \mid c$ (types)

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

$v ::= \ldots \mid \mathsf{rec}\ f\ x = e \mid \Lambda e \mid \mathsf{pack}\ v$ (values)

$e ::= \ldots \mid \mathsf{hash}\ e$ (expressions)

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

$$\kappa ::= \star \mid \kappa \Rightarrow \kappa$$ (kinds)

$$\tau ::= \alpha \mid \lambda\alpha : \kappa \,.\, \tau \mid \tau\ \tau \mid c$$ (types)

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

$$v ::= \ldots \mid \mathsf{rec}\ f\ x = e \mid \Lambda e \mid \mathsf{pack}\ v$$ (values)

$$e ::= \ldots \mid \mathsf{hash}\ e$$ (expressions)

We write, e.g., $\forall \alpha : \kappa \,.\, \tau$ to mean $\forall_\kappa\ (\lambda\alpha : \kappa \,.\, \tau)$ and $\tau_1 \times \tau_2$ for $\times\ \tau_1\ \tau_2$

Authentikit in $F^{ref}_{\omega,\mu}$

```
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
```

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

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

“F-ing” the module

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 $\mathsf{hashed}(s)$ such that

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

and

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

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

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:

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

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

Types:

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

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:

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

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

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:

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

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

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

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:

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

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

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

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

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:

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

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

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

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

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

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

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

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
```

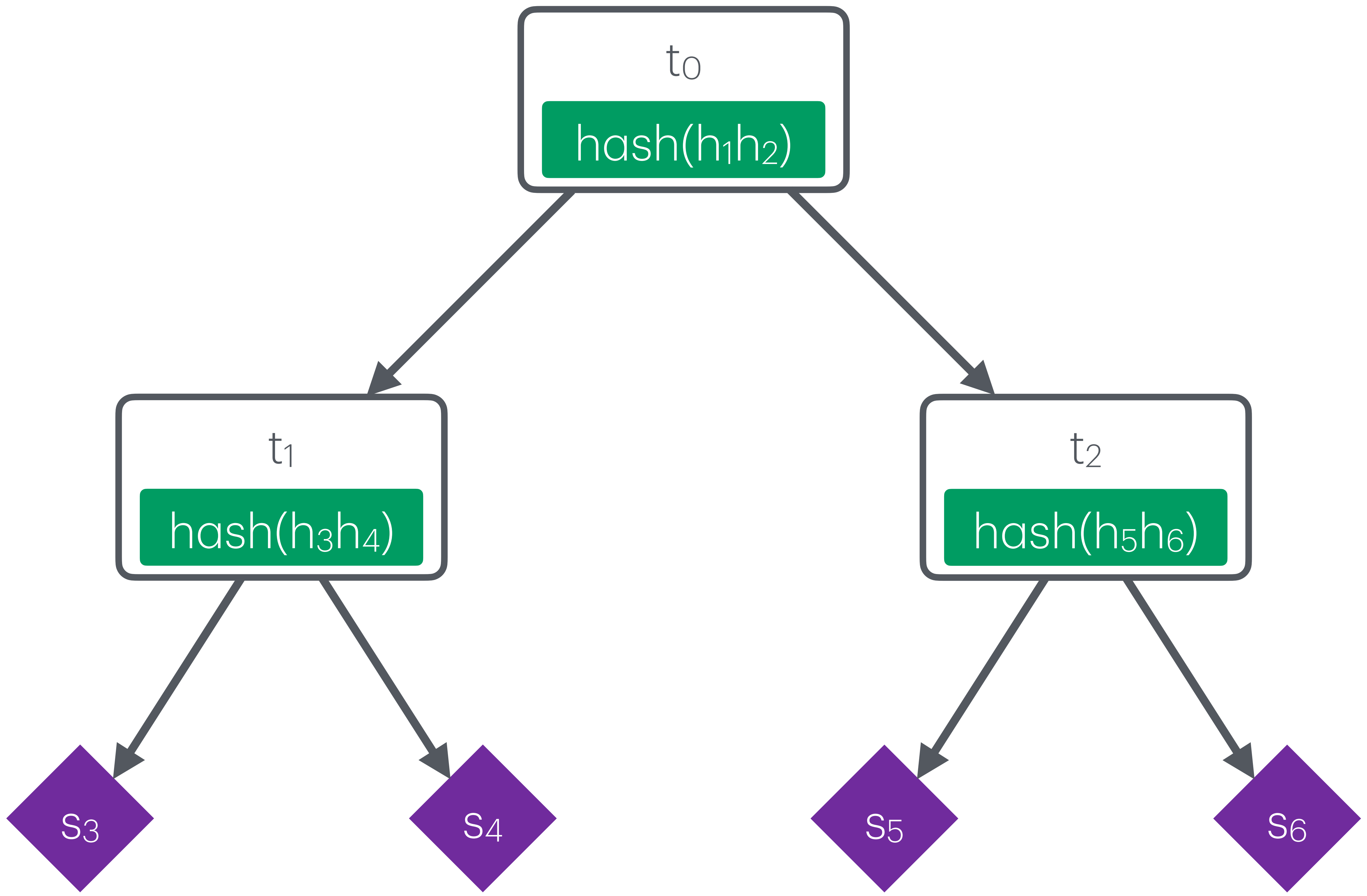
Manual client proofs

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:

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



Ablation study