NYU

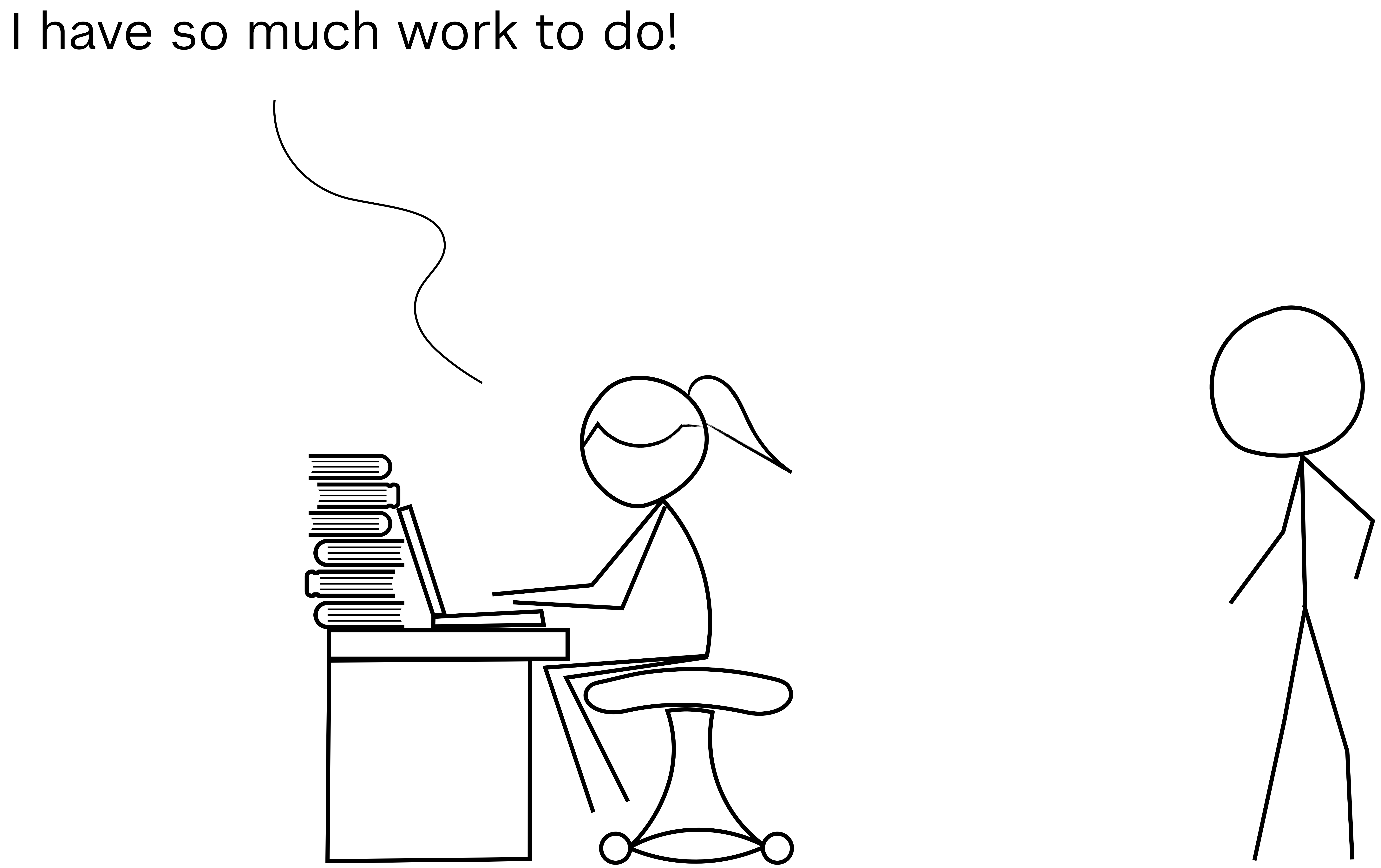
Logical Relations for Formally Verified

Authenticated Data Structures

Simon Oddershede Gregersen

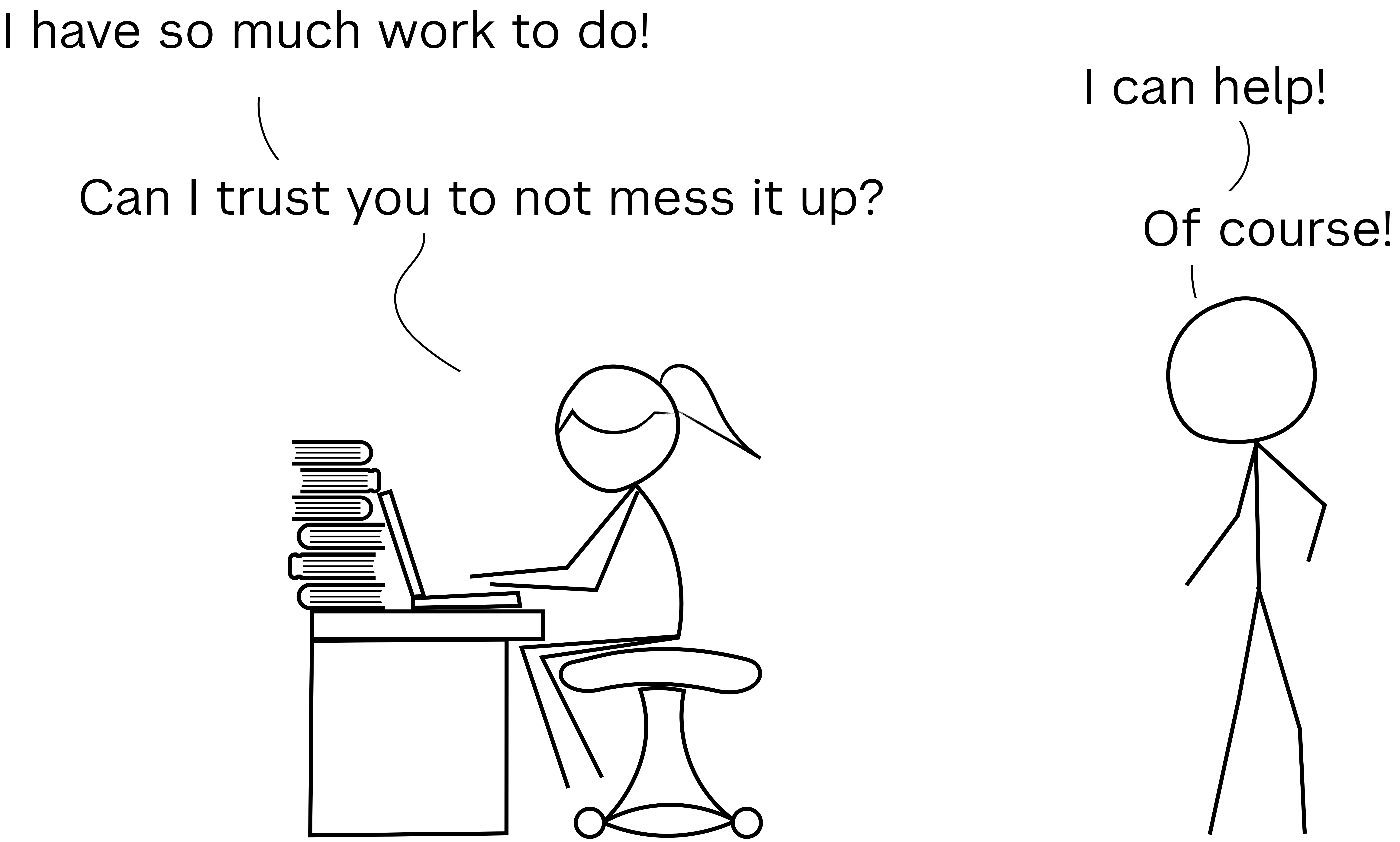
joint work with Chaitanya Agarwal and Joseph Tassarotti


I have so much work to do!


I have so much work to do!


I have so much work to do!
I can help!


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


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

How can Alice securely outsource work to Bob?

How can Alice securely outsource work to Bob?

The operations of an **authenticated data structure** can be carried out by Bob, but (efficiently) verified by, e.g., Alice!

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

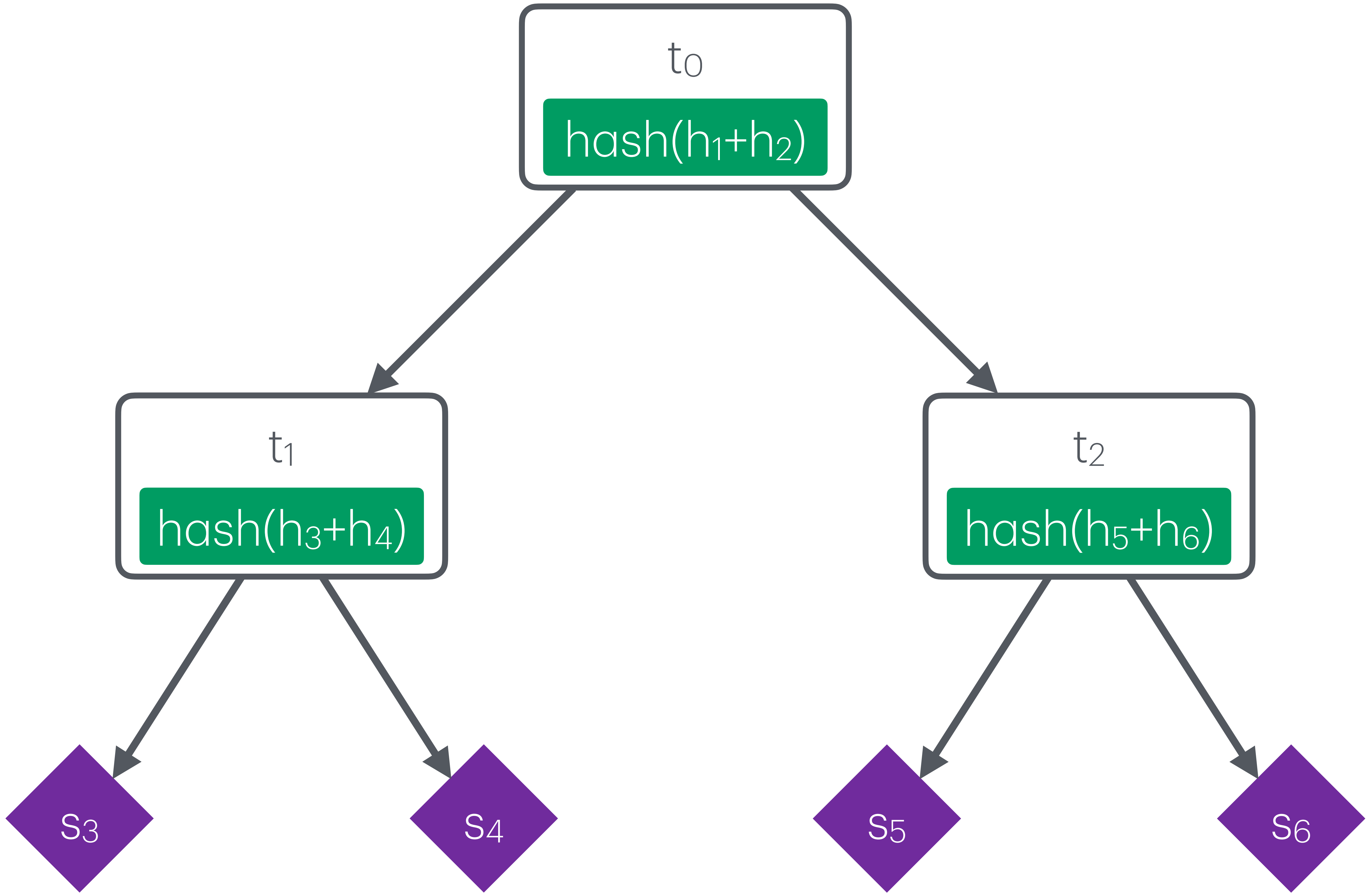
How can Alice securely outsource work to Bob?

The operations of an **authenticated data structure** can be carried out by Bob, but (efficiently) verified by, e.g., Alice!

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

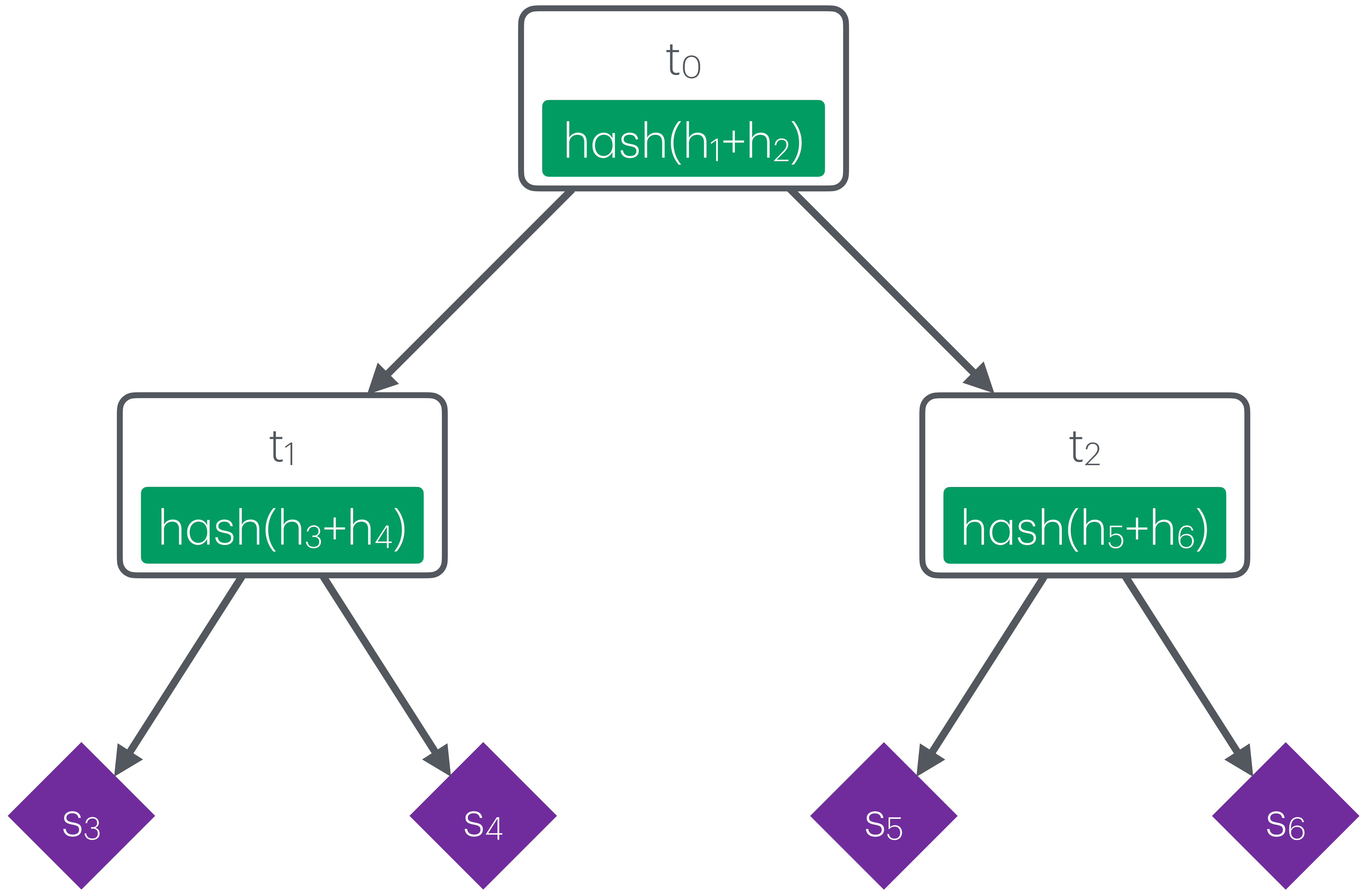
ADSs allow outsourcing data storage and processing tasks to untrusted servers without loss of integrity.

Example: Merkle Tree

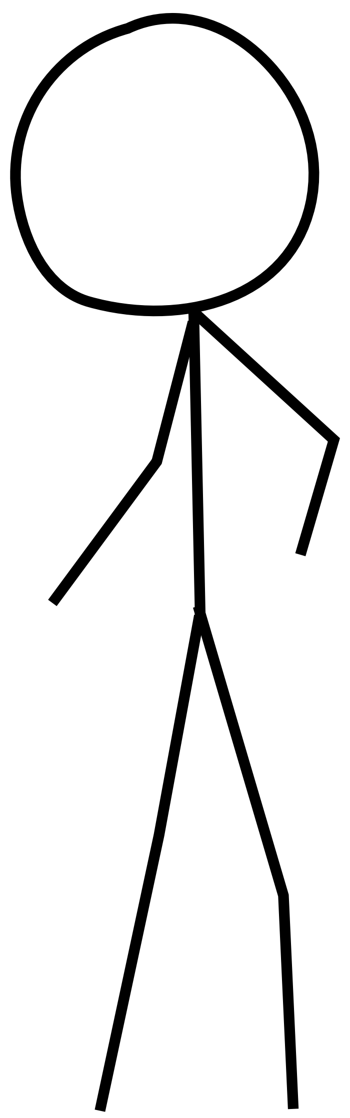


where h_i denotes the hash of t_i/s_i

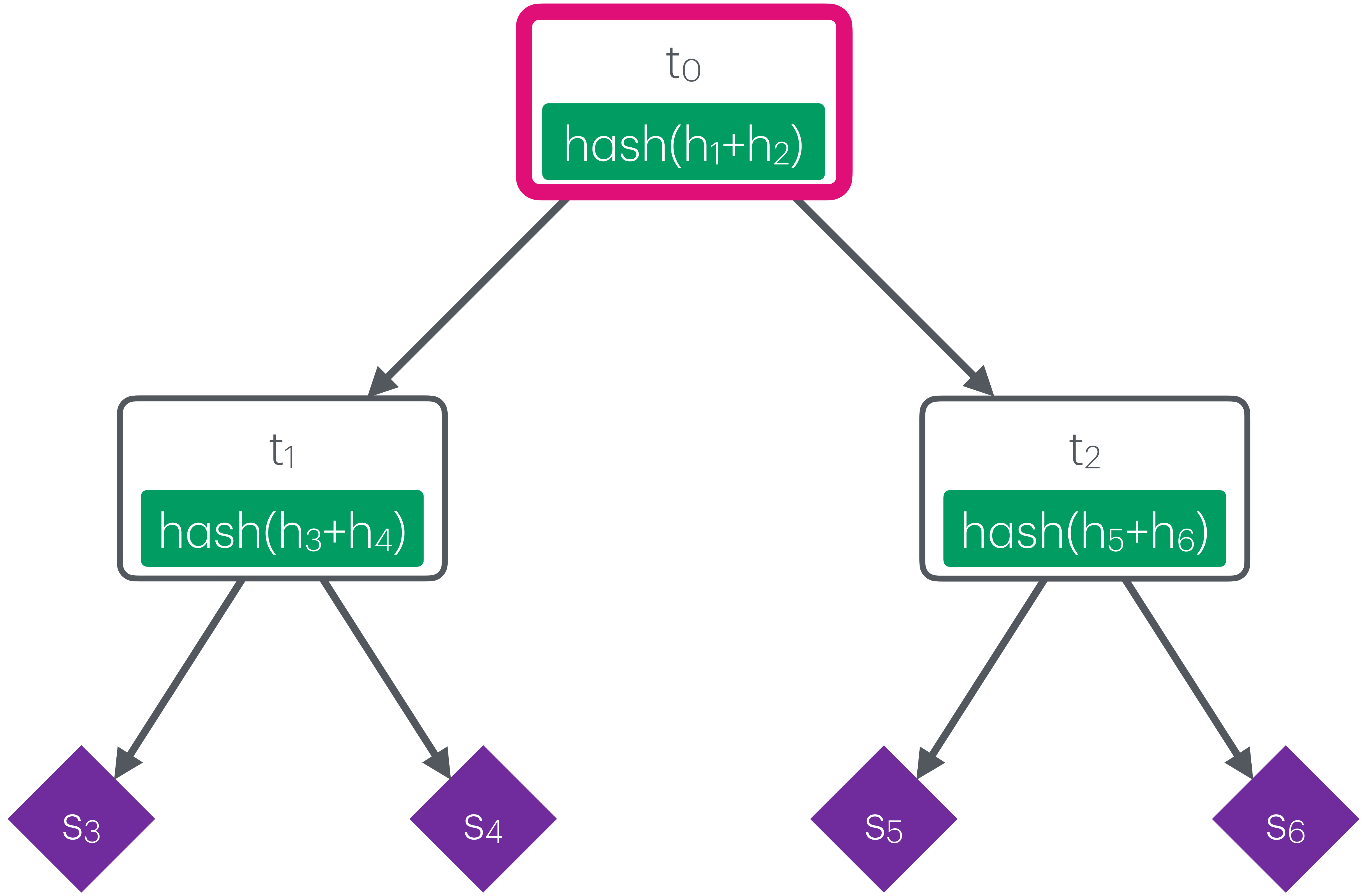
Example: Merkle Tree (Prover)



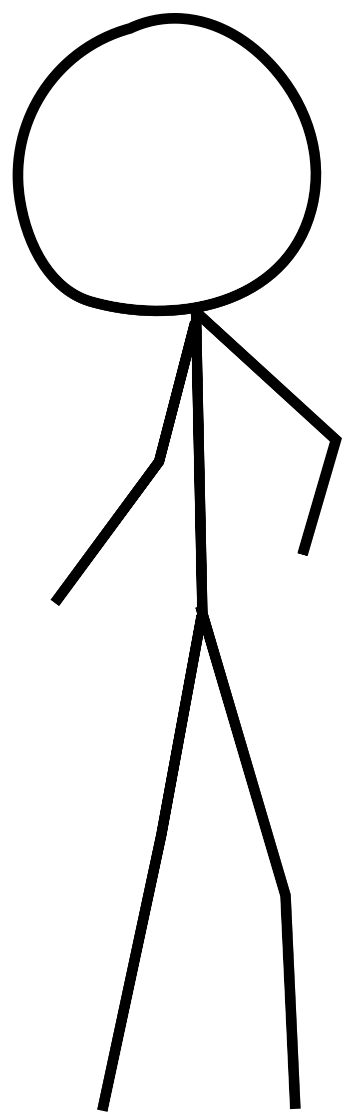
lookup([R, L], t_0) =



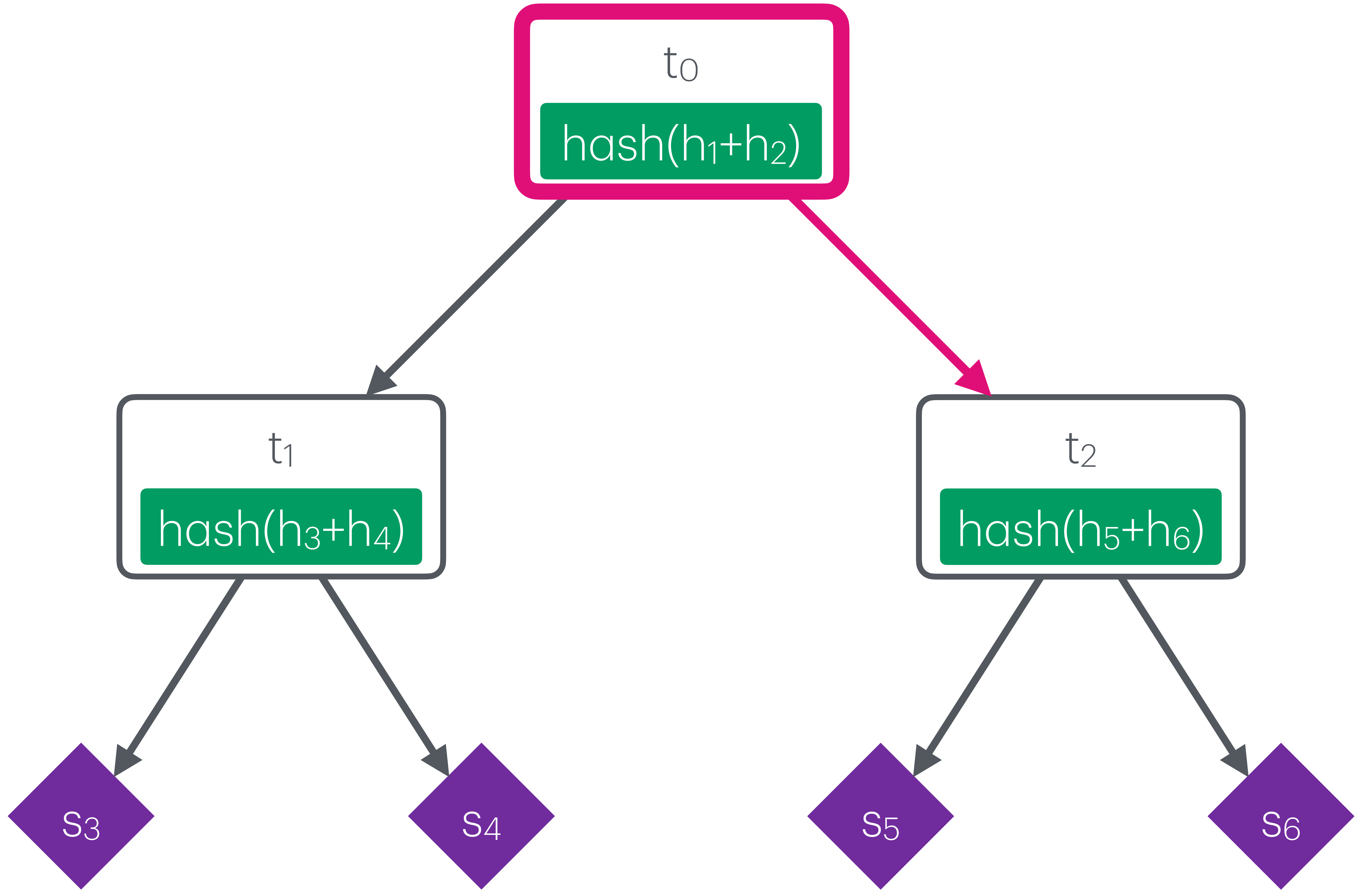
Example: Merkle Tree (Prover)



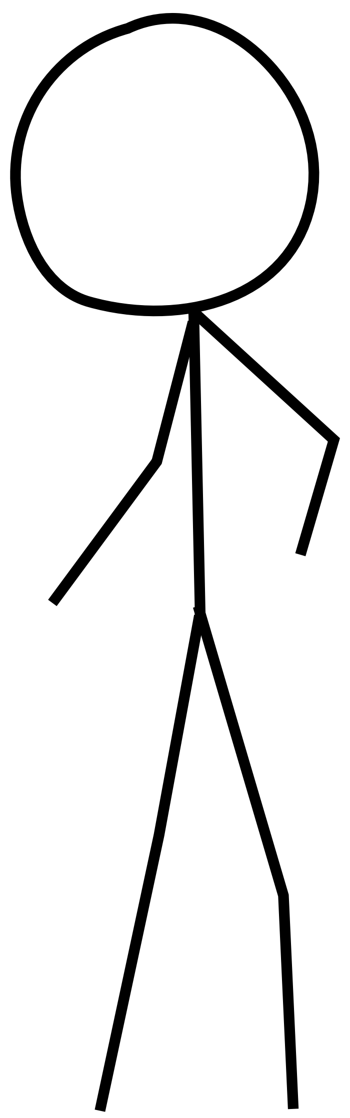
lookup([R, L], t_0) =



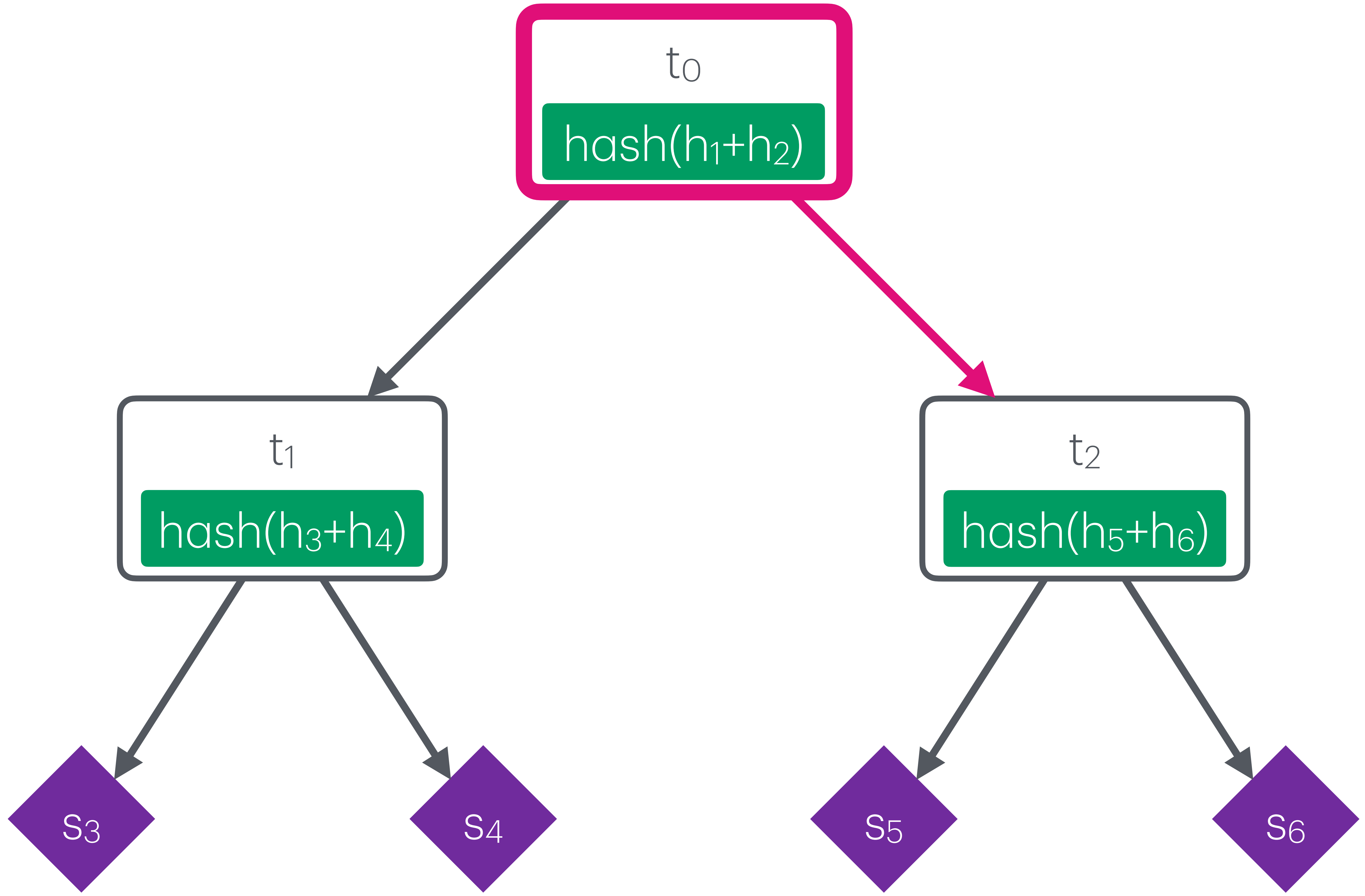
Example: Merkle Tree (Prover)



lookup([R, L], t_0) =

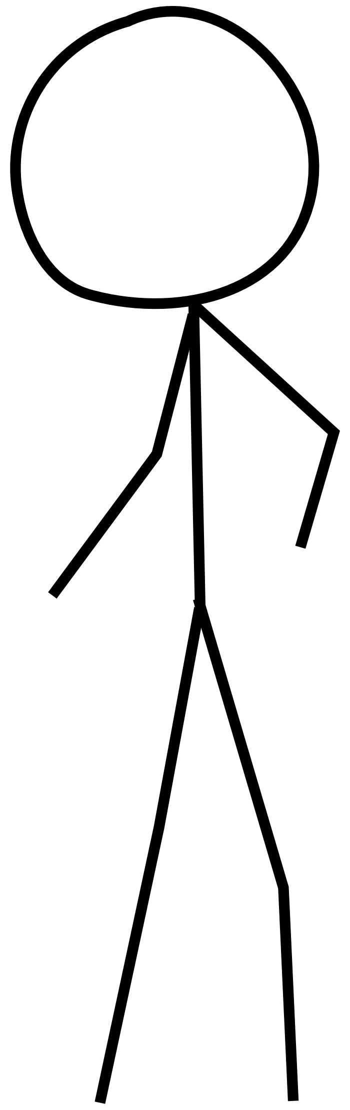


Example: Merkle Tree (Prover)

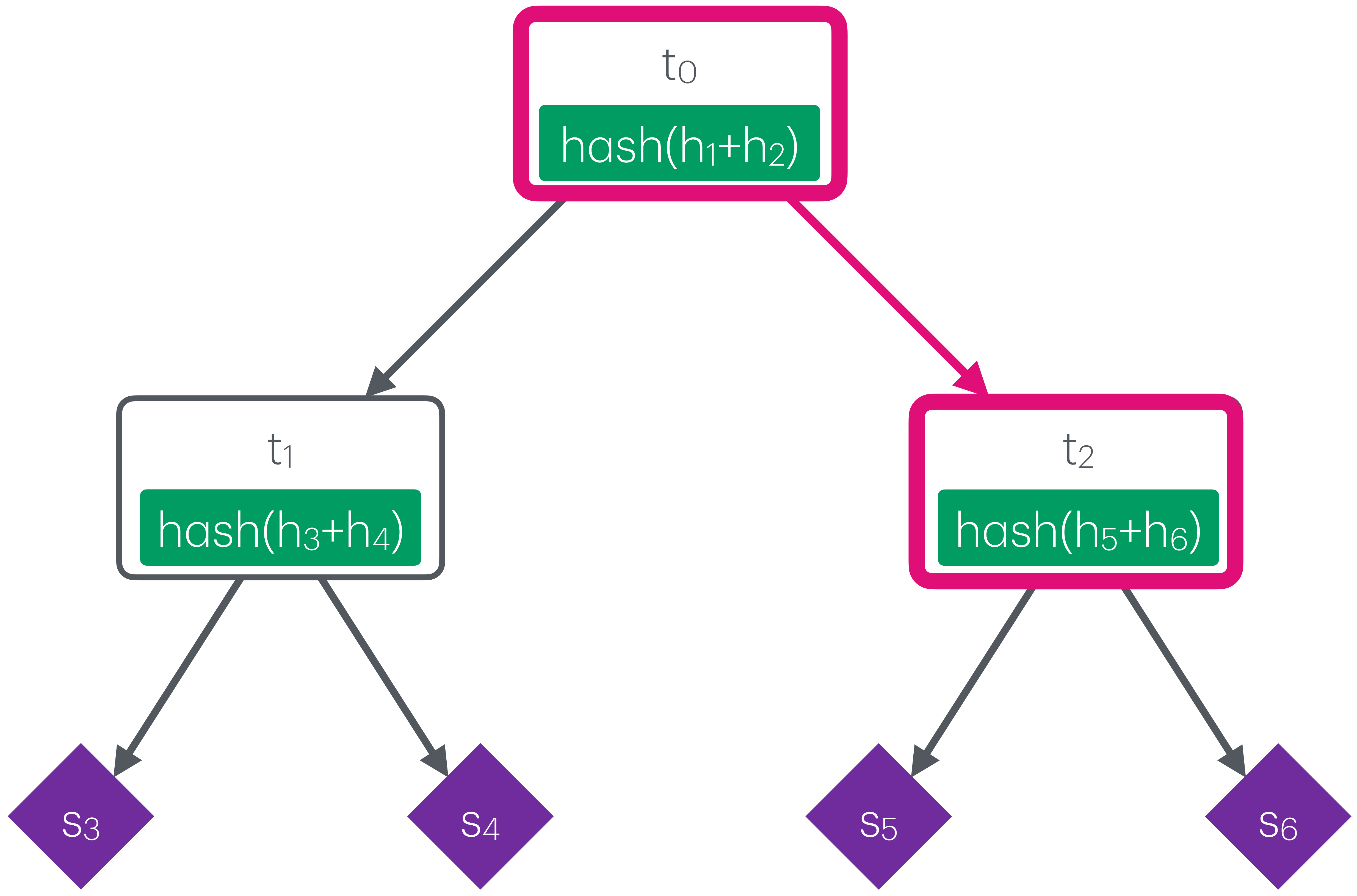


lookup([R, L], t_0) =

([h_1

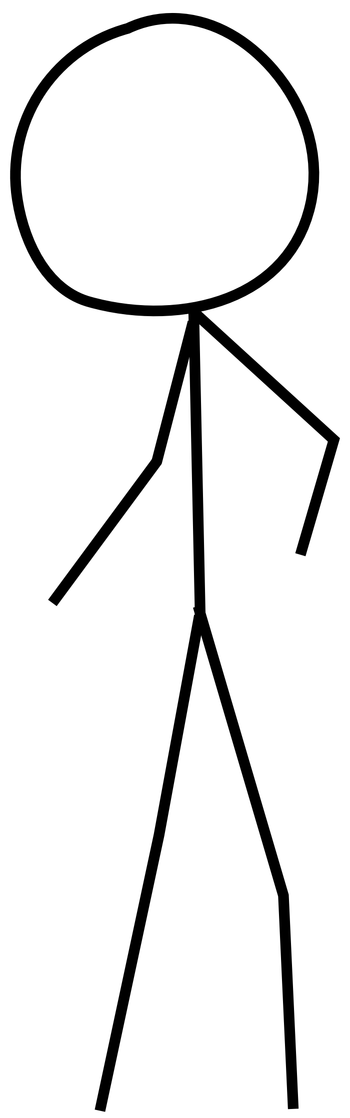


Example: Merkle Tree (Prover)

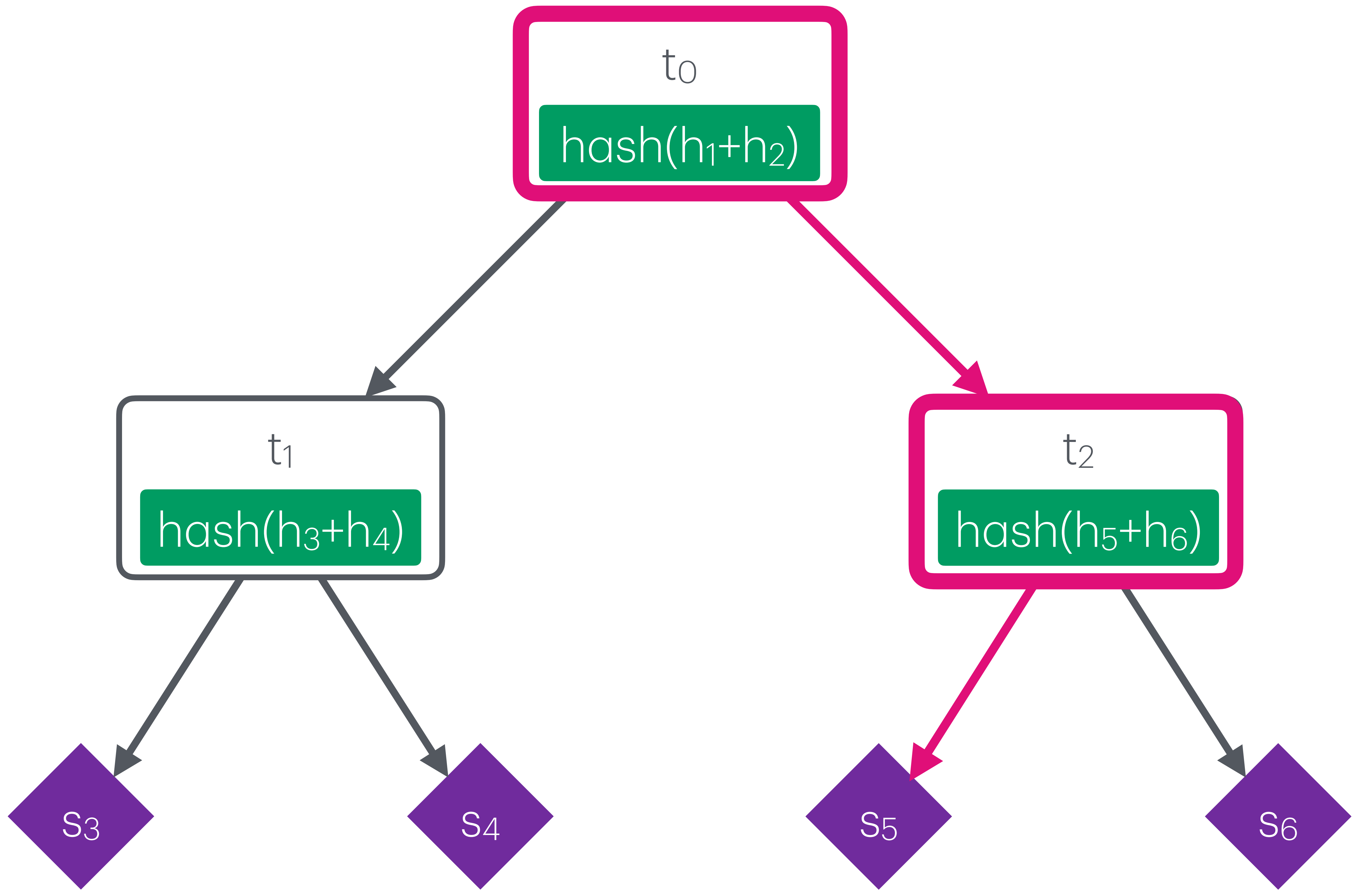


lookup([R, L], t_0) =

([h_1

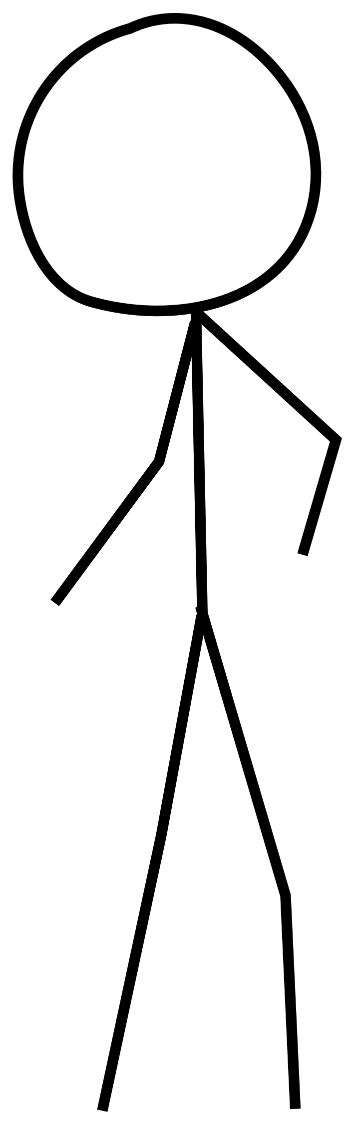


Example: Merkle Tree (Prover)

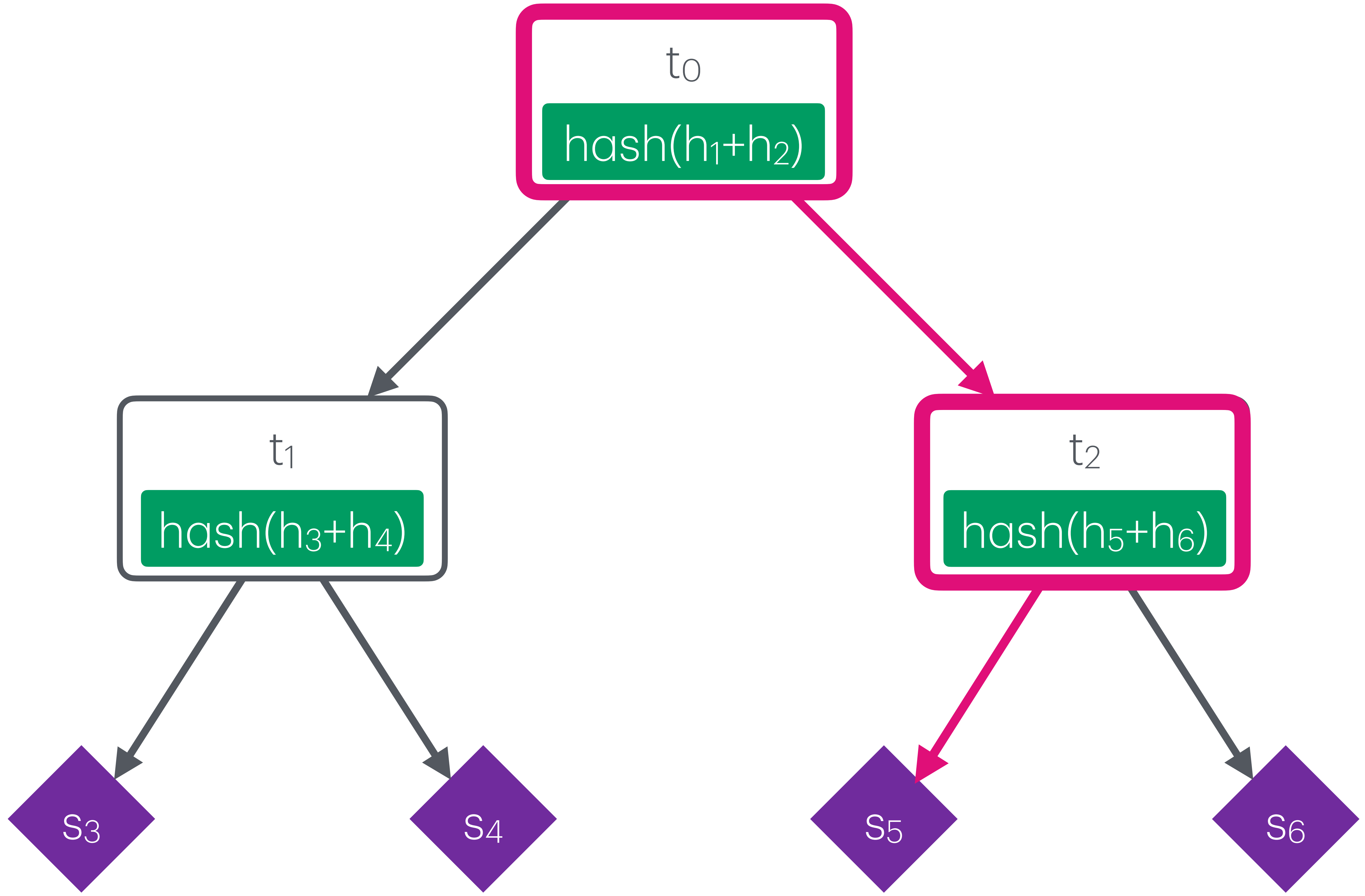


lookup([R, L], t_0) =

([h_1

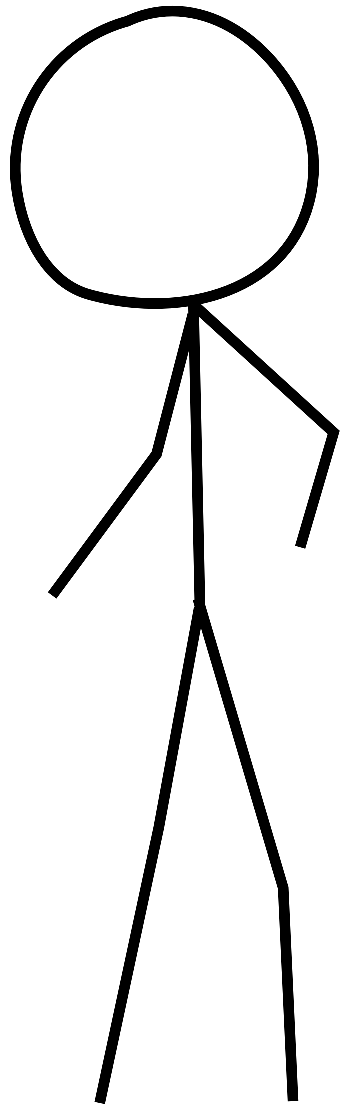


Example: Merkle Tree (Prover)

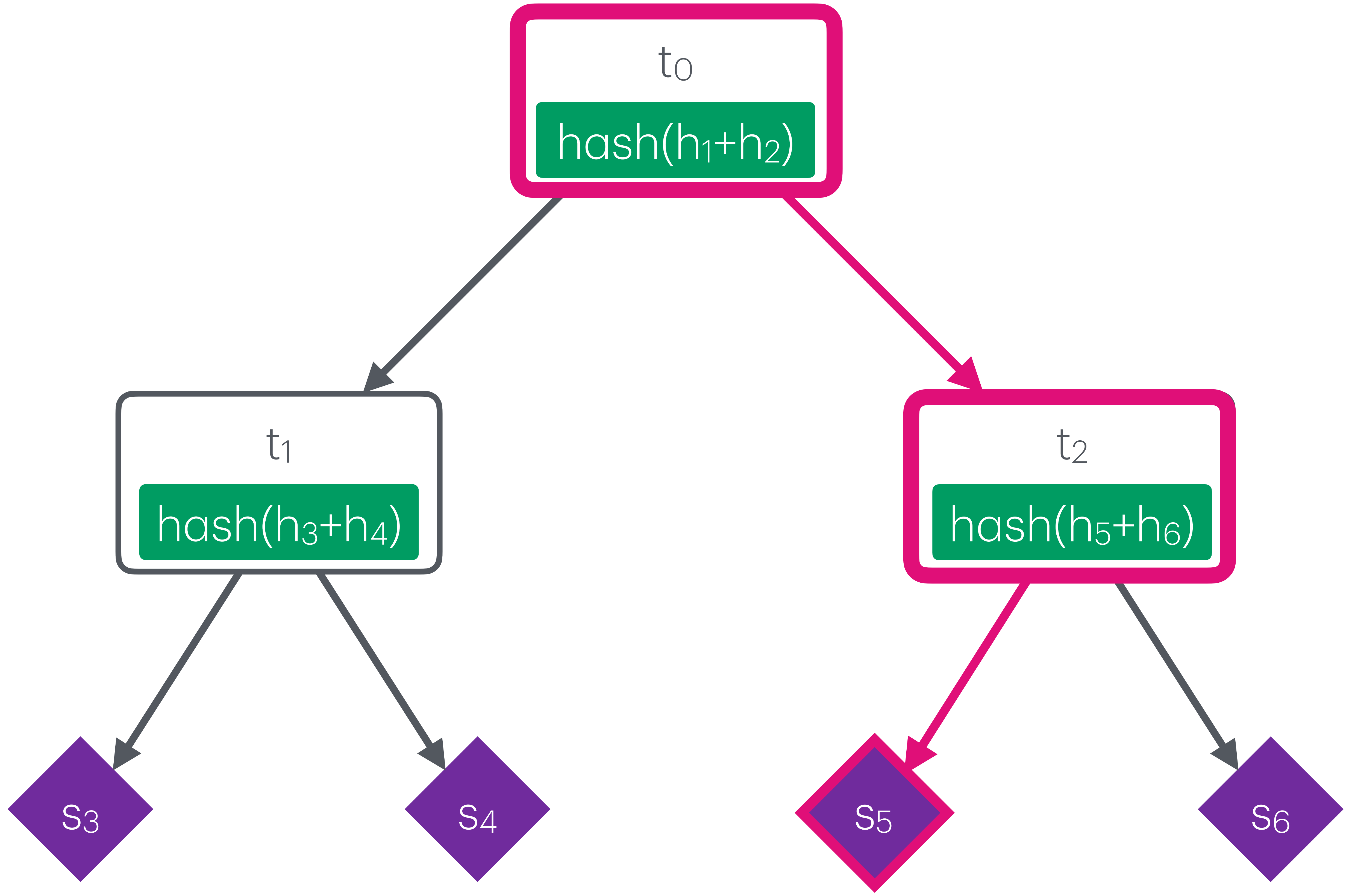


lookup([R, L], t_0) =

([h_1, h_6

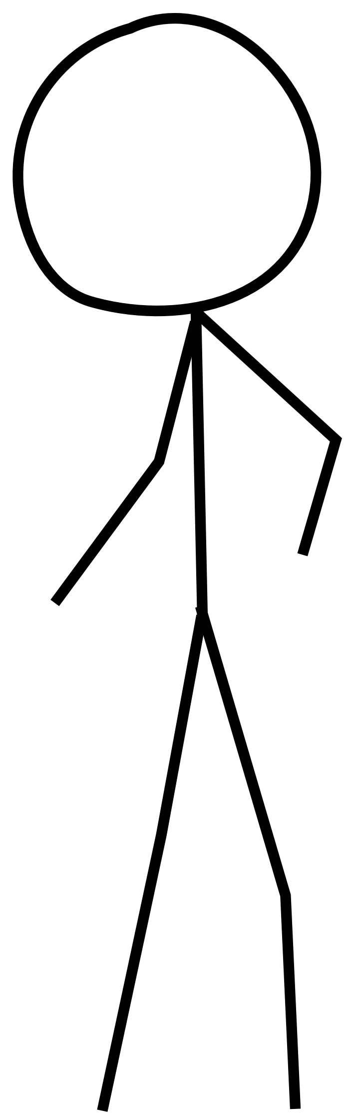


Example: Merkle Tree (Prover)

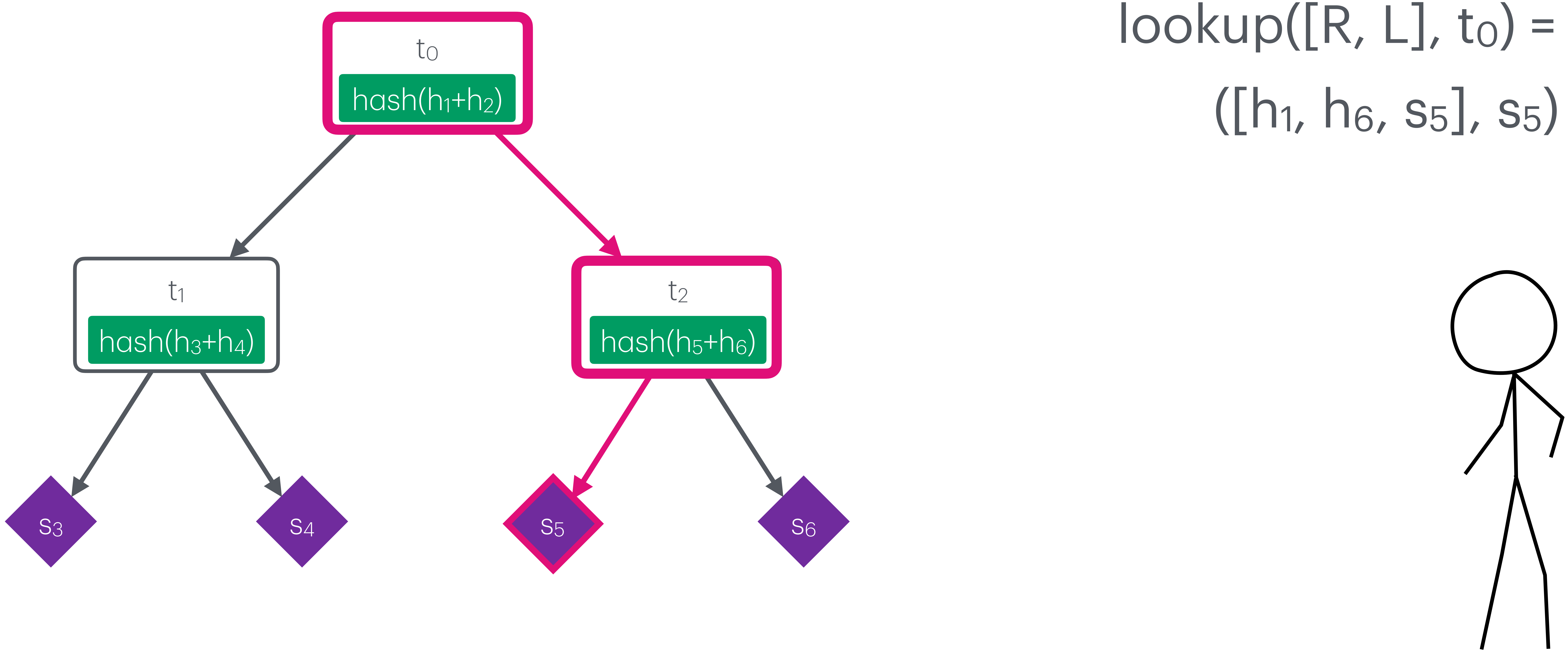


lookup([R, L], t_0) =

([h_1, h_6



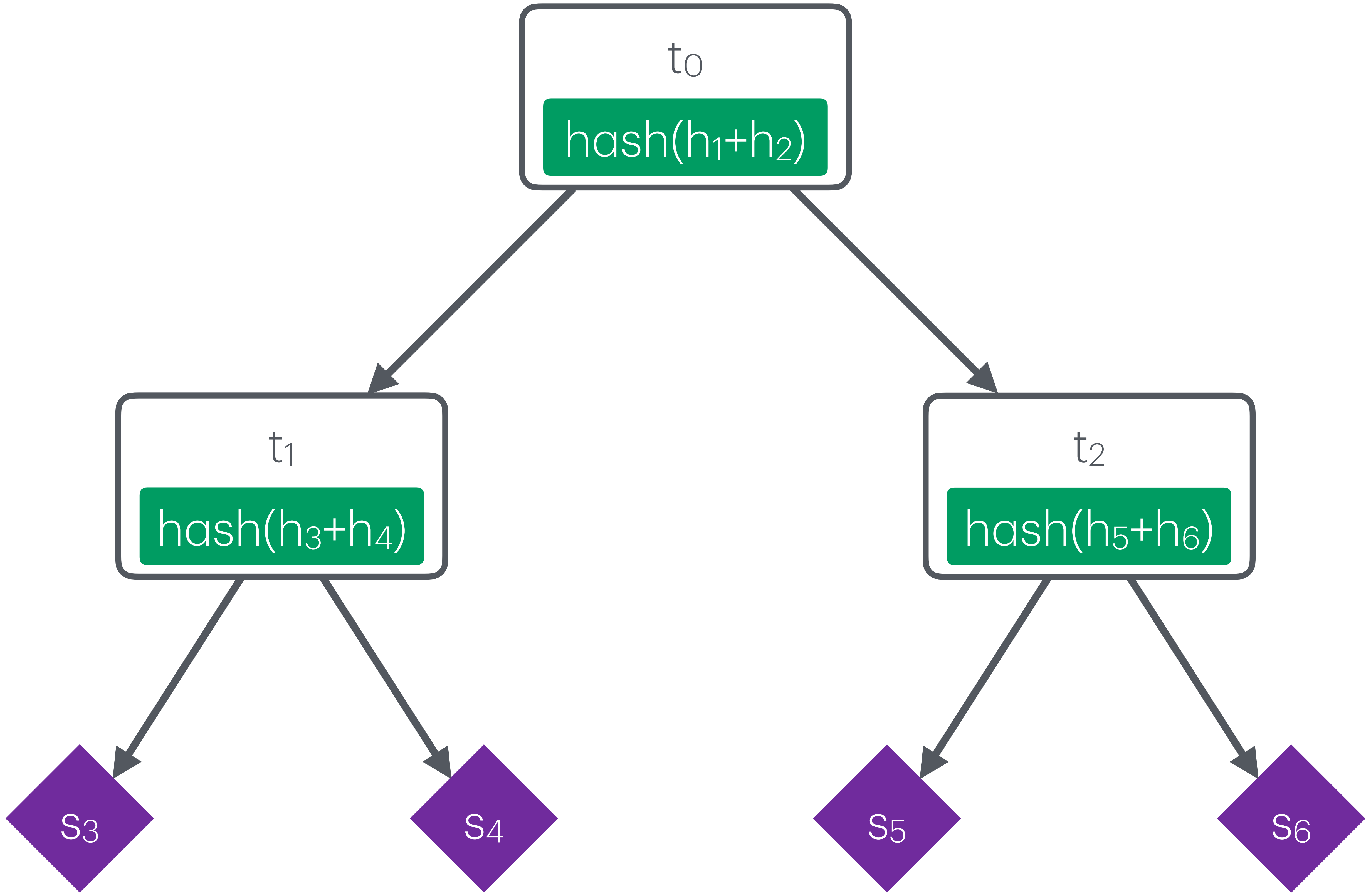
Example: Merkle Tree (Prover)



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

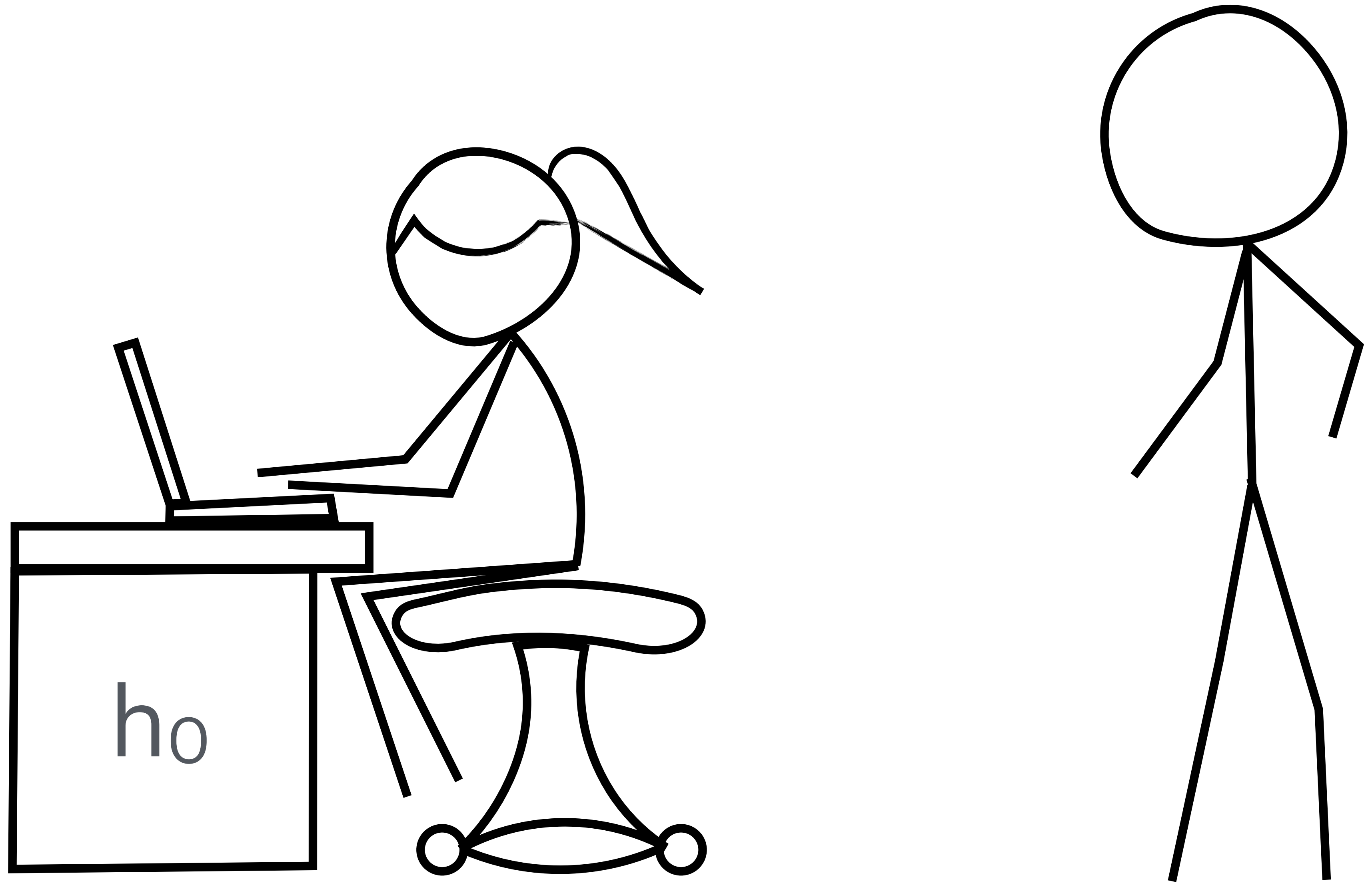
$([h_1, h_6, s_5], s_5)$

Example: Merkle Tree (Verifier)



lookup([R, L], t_0) =

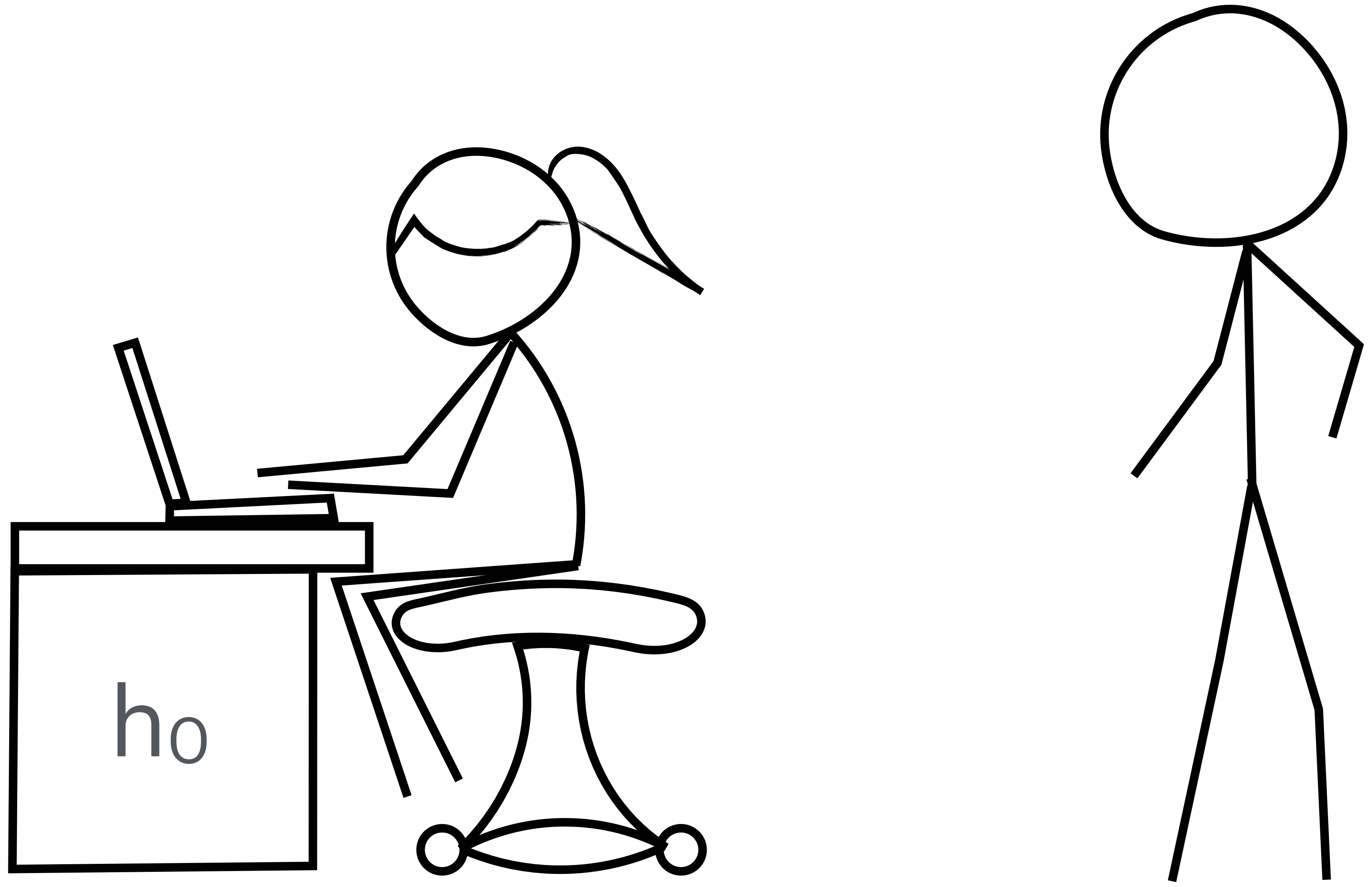
([h_1, h_6, s_5], s_5)



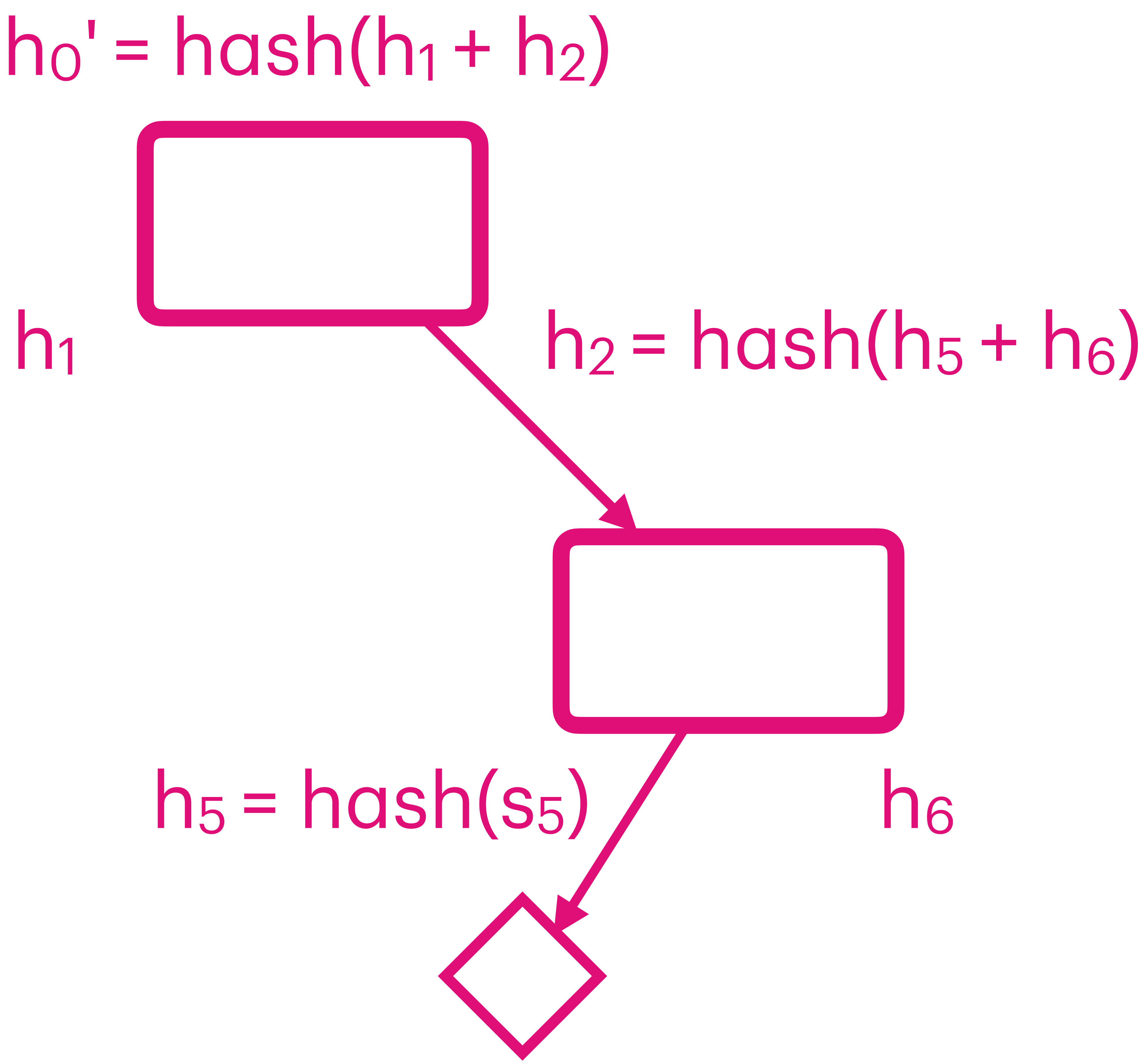
Example: Merkle Tree (Verifier)

lookup([R, L], t_0) =

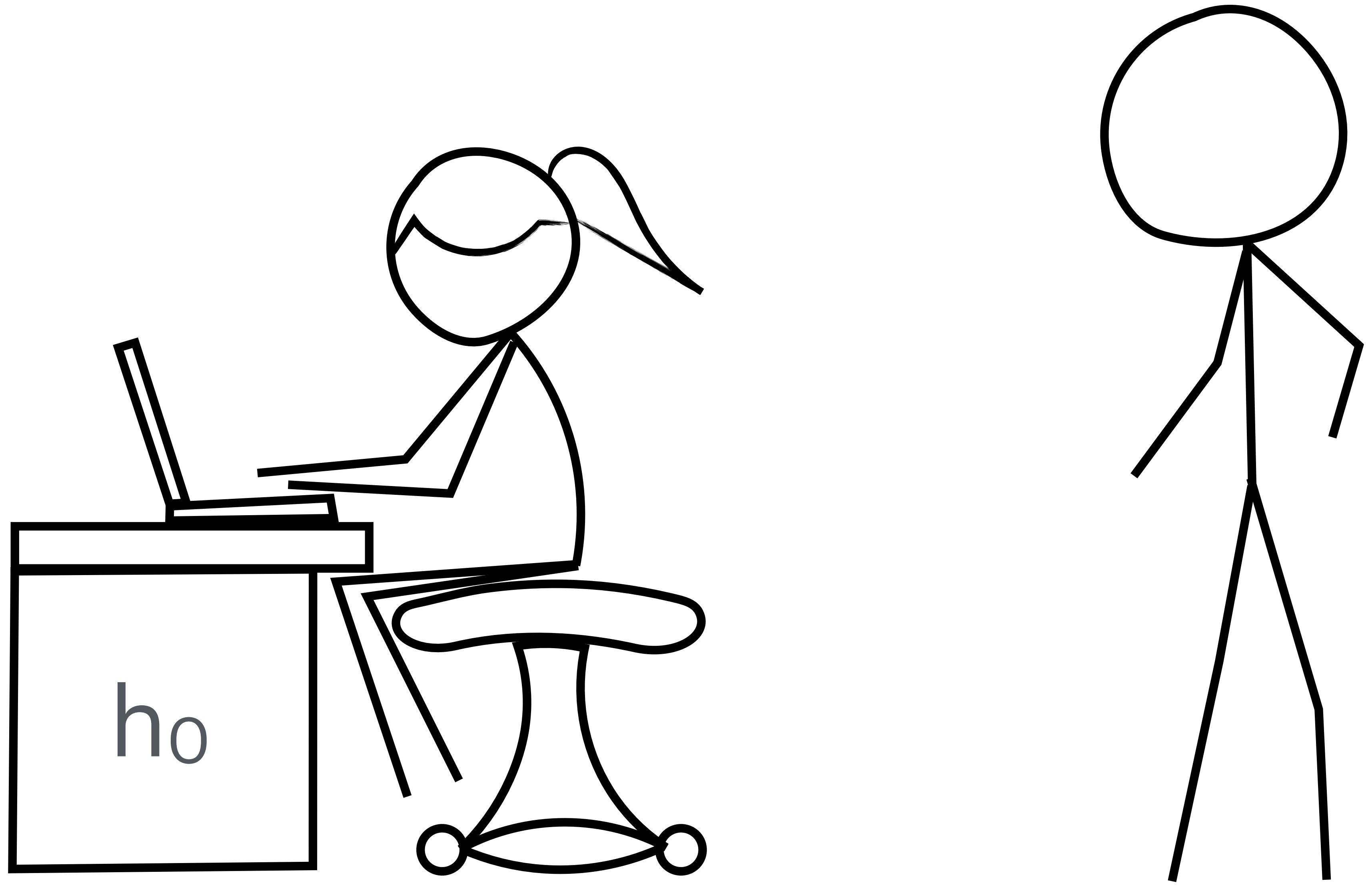
([h_1, h_6, s_5], s_5)



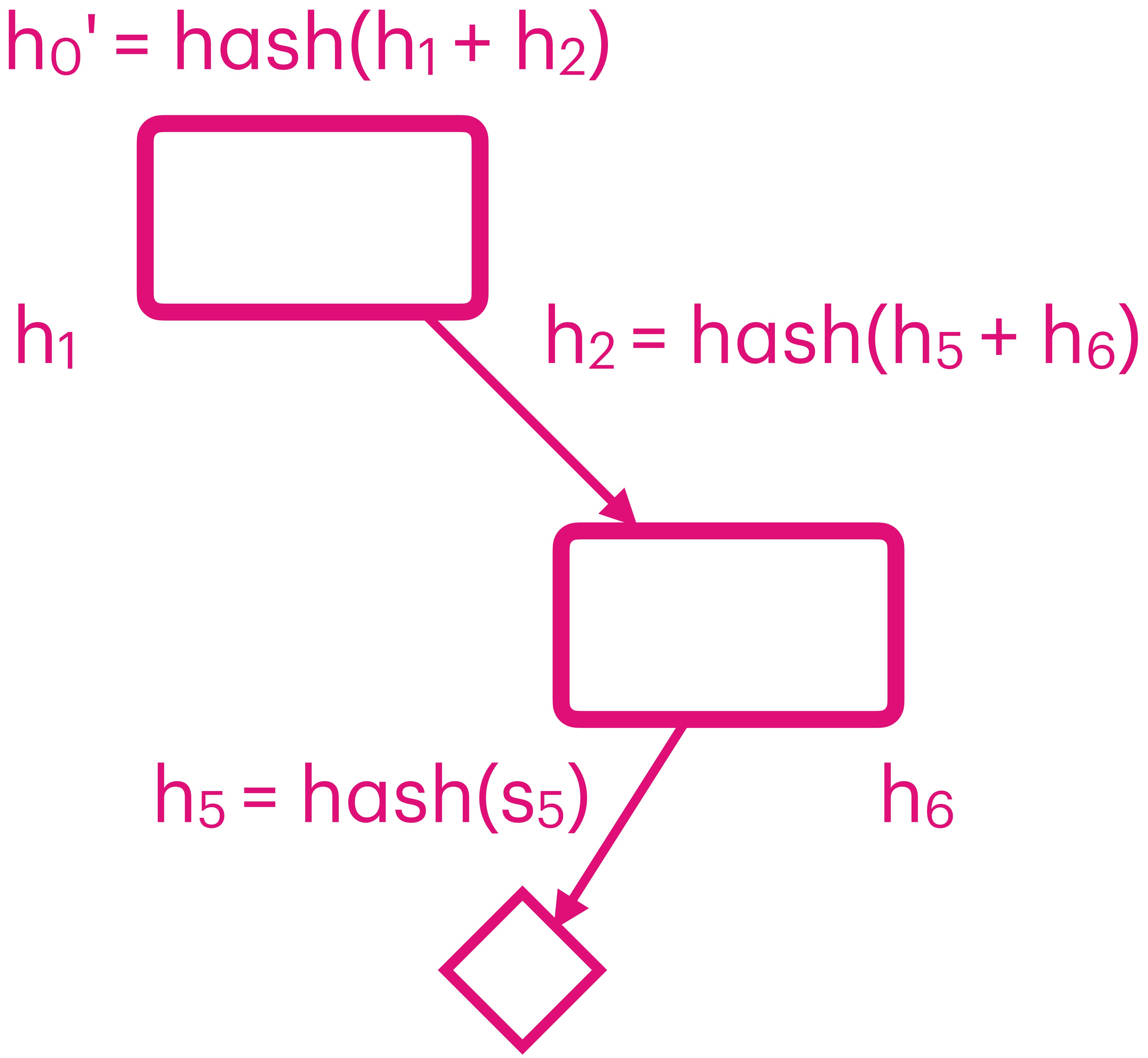
Example: Merkle Tree (Verifier)



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

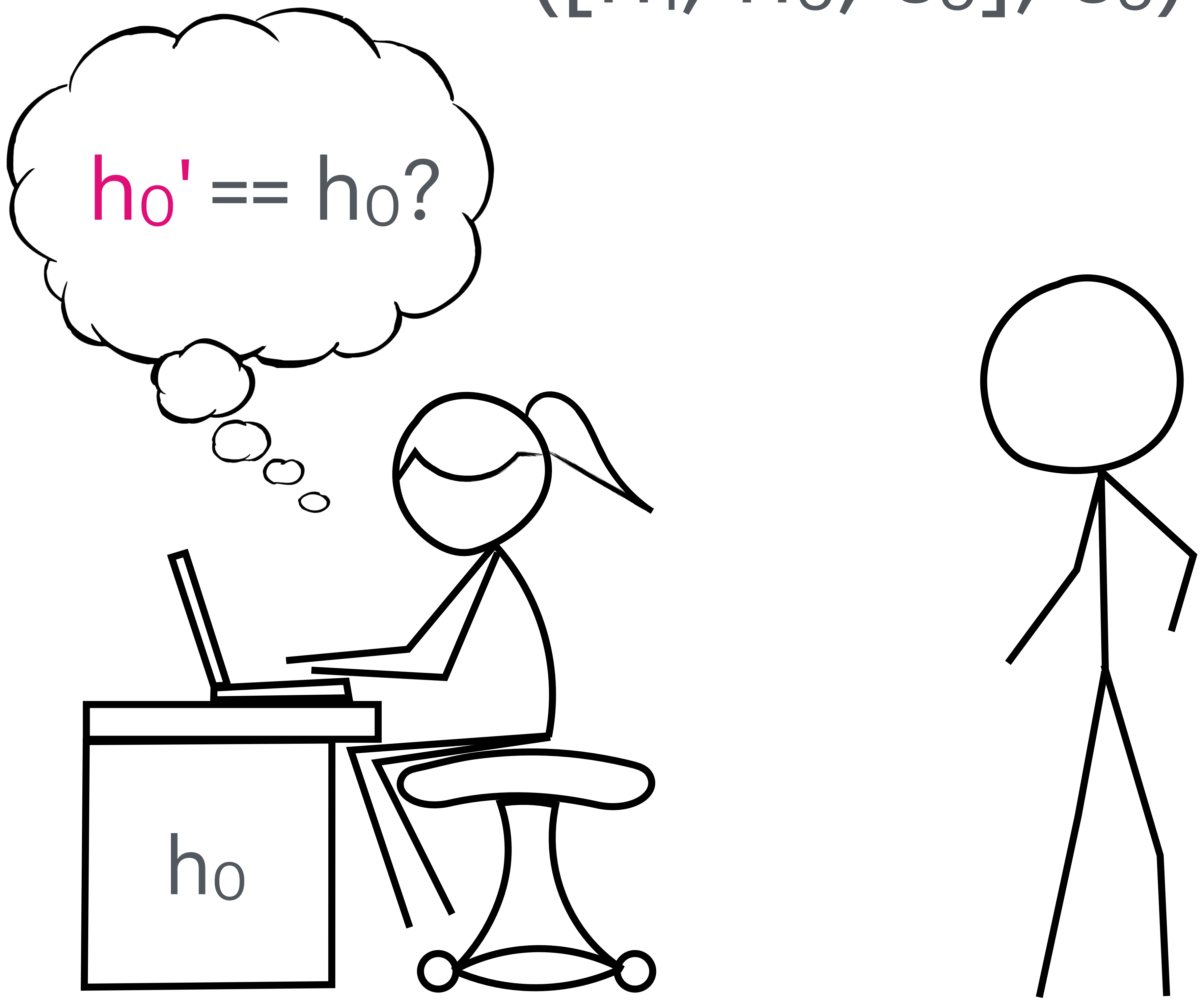


Example: Merkle Tree (Verifier)



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

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



Use cases

- **Certificate transparency**

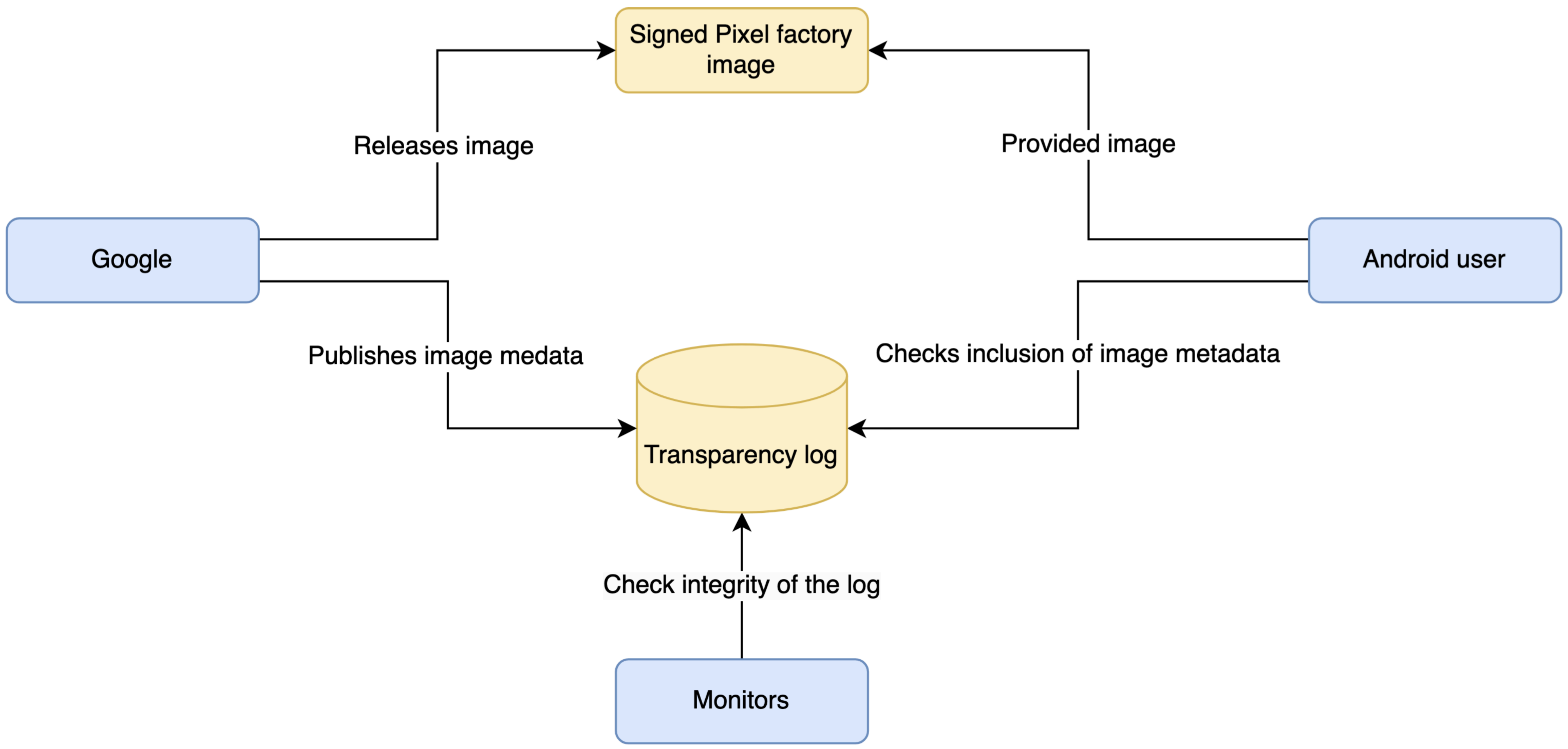
  Google Chrome (2015), Cloudflare (2018), Let's Encrypt (2019), Firefox (2025)

- **Key transparency**

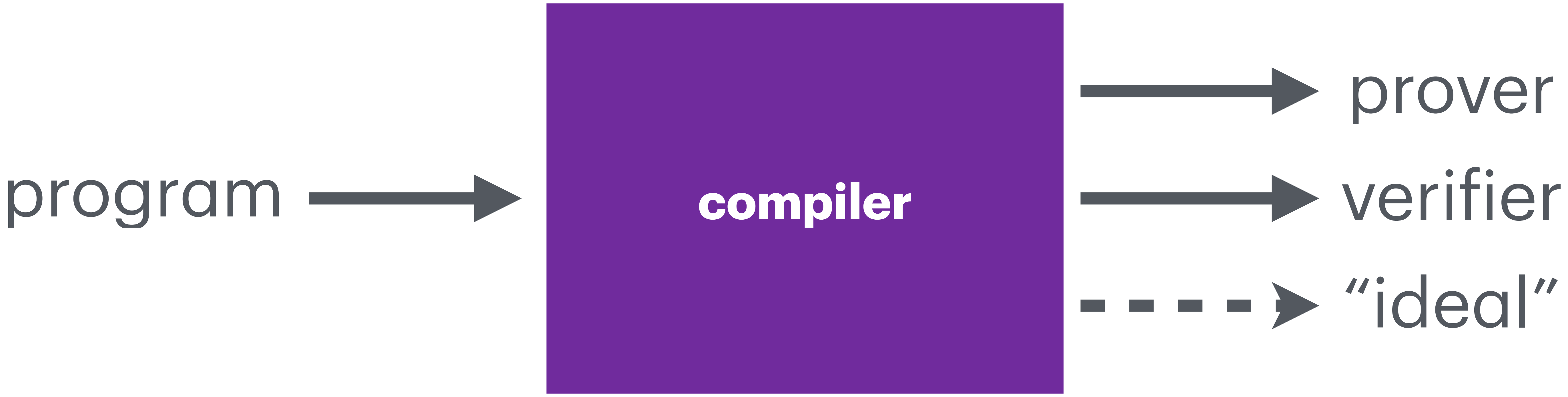
  WhatsApp (2023), Signal (???)

- **Binary transparency**

  Pixel Binaries, Go modules



Miller et al. realized that the prover and verifier can be **compiled** from a single implementation.



Authenticated Data Structures, Generically

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

Abstract

An authenticated data structure (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. This is done by having the prover produce a compact proof that the verifier can check along with each operation's result. ADSs thus support outsourcing data maintenance and processing tasks to untrusted servers without loss of integrity. Past work on ADSs has focused on particular data structures (or limited classes of data structures), one at a time, often with support only for particular operations.

This paper presents a generic method, using a simple extension to a ML-like functional programming language we call λ• (lambda-auth), with which one can program authenticated operations over any data structure defined by standard type constructors, including recursive types, sums, and products. The programmer writes the data structure largely as usual and it is compiled to code to be run by the prover and verifier. Using a formalization of λ• we prove that all well-typed λ• programs result in code that is secure under the standard cryptographic assumption of collision-resistant hash functions. We have implemented λ• as an extension to the OCaml compiler, and have used it to produce authenticated versions of many interesting data structures including binary search trees, red-black+ trees, skip lists, and more. Performance experiments show that our approach is efficient, giving up little compared to the hand-optimized data structures developed previously.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

General Terms Security, Programming Languages, Cryptography

1. Introduction

Suppose data provider would like to allow third parties to mirror its data, providing a query interface over it to clients. The data provider wants to assure clients that the mirrors will answer queries over the data truthfully, even if they (or another party that compromises a mirror) have an incentive to lie. As examples, the data provider might be providing stock market data, a certificate revocation list, the Tor relay list, or the state of the current Bitcoin ledger [22].

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535851

Such a scenario can be supported using *authenticated data structures* (ADS) [5, 24, 31]. ADS computations involve two roles, the *prover* and the *verifier*. The mirror plays the role of the prover, storing the data of interest and answering queries about it. The client plays the role of the verifier, posing queries to the prover and verifying that the returned results are authentic. At any point in time, the verifier holds only a short *digest* that can be viewed as summarizing the current contents of the data; an authentic copy of the digest is provided by the data owner. When the verifier sends the prover a query, the prover computes the result and returns it along with a *proof* that the returned result is correct; both the proof and the time to produce it are linear in the time to compute the query result. The verifier can attempt to verify the proof (in time linear in the size of the proof) using its current digest, and will accept the returned result only if the proof verifies. If the verifier is also the data provider, the verifier may also update its data stored at the prover; in this case, the result is an updated digest and the proof shows that this updated digest was computed correctly. ADS computations have two properties. *Correctness* implies that when both parties execute the protocol correctly, the proofs given by the prover verify correctly and the verifier always receives the correct result. *Security*[1] implies that a computationally bounded, malicious prover cannot fool the verifier into accepting an incorrect result.

Authenticated data structures can be traced back to Merkle [18]; the well-known *Merkle hash tree* can be viewed as providing an authenticated version of a bounded-length array. More recently, authenticated versions of data structures as diverse as sets [23, 27], dictionaries [1, 12], range trees [16], graphs [13], skip lists [11, 12], B-trees [21], hash trees [26], and more [15] have been proposed. In each of these cases, the design of the data structure, the supporting operations, and how they can be proved authentic have been reconsidered from scratch, involving a new, potentially tricky proof of security. Arguably, this state of affairs has hindered the advancement of new data-structure designs as previous ideas are not easily reused or reapplied. We believe that ADSs will make their way into systems more often if they become easier to build.

This paper presents λ• (pronounced "lambda auth"), a language for programming authenticated data structures. λ• represents the first *generic*, language-based approach to building dynamic authenticated data structures with provable guarantees. The key observation underlying λ•'s design is that, whatever the data structure or operation, the computations performed by the prover and verifier can be made structurally the same: the prover constructs the proof at key points when executing a query, and the verifier checks a proof by using it to "replay" the query, checking at each key point that the computation is self-consistent.

λ• implements this idea using what we call *authenticated types*, written $\bullet\tau$, with coercions *auth* and *unauth* for introducing and eliminating values of an authenticated type. Using standard func-

[1] This property is sometimes called *soundness* but we eschew this term to avoid confusion with its standard usage in programming languages.

411

Miller et. al's approach

OCaml is extended with three new primitives:

- authenticated types $\bullet\, \tau$
- auth : $\forall \alpha\, .\, \alpha \rightarrow \bullet\, \alpha$
- unauth : $\forall \alpha .\, \bullet\, \alpha \rightarrow \alpha$

Authenticated Data Structures, Generically

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

Abstract

An authenticated data structure (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. This is done by having the prover produce a compact proof that the verifier can check along with each operation's result. ADSs thus support outsourcing data maintenance and processing tasks to untrusted servers without loss of integrity. Past work on ADSs has focused on particular data structures (or limited classes of data structures), one at a time, often with support only for particular operations.

This paper presents a generic method, using a simple extension to a ML-like functional programming language we call λ• (lambda-auth), with which one can program authenticated operations over any data structure defined by standard type constructors, including recursive types, sums, and products. The programmer writes the data structure largely as usual and it is compiled to code to be run by the prover and verifier. Using a formalization of λ• we prove that all well-typed λ• programs result in code that is secure under the standard cryptographic assumption of collision-resistant hash functions. We have implemented λ• as an extension to the OCaml compiler, and have used it to produce authenticated versions of many interesting data structures including binary search trees, red-black+ trees, skip lists, and more. Performance experiments show that our approach is efficient, giving up little compared to the hand-optimized data structures developed previously.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

General Terms Security, Programming Languages, Cryptography

1. Introduction

Suppose data provider would like to allow third parties to mirror its data, providing a query interface over it to clients. The data provider wants to assure clients that the mirrors will answer queries over the data truthfully, even if they (or another party that compromises a mirror) have an incentive to lie. As examples, the data provider might be providing stock market data, a certificate revocation list, the Tor relay list, or the state of the current Bitcoin ledger [22].

Such a scenario can be supported using *authenticated data structures* (ADS) [5, 24, 31]. ADS computations involve two roles, the *prover* and the *verifier*. The mirror plays the role of the prover, storing the data of interest and answering queries about it. The client plays the role of the verifier, posing queries to the prover and verifying that the returned results are authentic. At any point in time, the verifier holds only a short *digest* that can be viewed as summarizing the current contents of the data; an authentic copy of the digest is provided by the data owner. When the verifier sends the prover a query, the prover computes the result and returns it along with a *proof* that the returned result is correct; both the proof and the time to produce it are linear in the time to compute the query result. The verifier can attempt to verify the proof (in time linear in the size of the proof) using its current digest, and will accept the returned result only if the proof verifies. If the verifier is also the data provider, the verifier may also update its data stored at the prover; in this case, the result is an updated digest and the proof shows that this updated digest was computed correctly. ADS computations have two properties. *Correctness* implies that when both parties execute the protocol correctly, the proofs given by the prover verify correctly and the verifier always receives the correct result. *Security*[1] implies that a computationally bounded, malicious prover cannot fool the verifier into accepting an incorrect result.

Authenticated data structures can be traced back to Merkle [18]; the well-known *Merkle hash tree* can be viewed as providing an authenticated version of a bounded-length array. More recently, authenticated versions of data structures as diverse as sets [23, 27], dictionaries [1, 12], range trees [16], graphs [13], skip lists [11, 12], B-trees [21], hash trees [26], and more [15] have been proposed. In each of these cases, the design of the data structure, the supporting operations, and how they can be proved authentic have been reconsidered from scratch, involving a new, potentially tricky proof of security. Arguably, this state of affairs has hindered the advancement of new data-structure designs as previous ideas are not easily reused or reapplied. We believe that ADSs will make their way into systems more often if they become easier to build.

This paper presents λ• (pronounced "lambda auth"), a language for programming authenticated data structures. λ• represents the first *generic*, language-based approach to building dynamic authenticated data structures with provable guarantees. The key observation underlying λ•'s design is that, whatever the data structure or operation, the computations performed by the prover and verifier can be made structurally the same: the prover constructs the proof at key points when executing a query, and the verifier checks a proof by using it to "replay" the query, checking at each key point that the computation is self-consistent.

λ• implements this idea using what we call *authenticated types*, written $\bullet\tau$, with coercions *auth* and *unauth* for introducing and eliminating values of an authenticated type. Using standard func-

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535851

[1] This property is sometimes called *soundness* but we eschew this term to avoid confusion with its standard usage in programming languages.

Miller et. al's approach

OCaml is extended with three new primitives:

- authenticated types $\bullet\,\tau$
- auth : $\forall \alpha\,.\,\alpha \rightarrow \bullet\,\alpha$
- unauth : $\forall \alpha.\,\bullet\,\alpha \rightarrow \alpha$

Authenticated Data Structures, Generically

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

Abstract

An authenticated data structure (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. This is done by having the prover produce a compact proof that the verifier can check along with each operation's result. ADSs thus support outsourcing data maintenance and processing tasks to untrusted servers without loss of integrity. Past work on ADSs has focused on particular data structures (or limited classes of data structures), one at a time, often with support only for particular operations.

This paper presents a generic method, using a simple extension to a ML-like functional programming language we call λ• (lambda-auth), with which one can program authenticated operations over any data structure defined by standard type constructors, including recursive types, sums, and products. The programmer writes the data structure largely as usual and it is compiled to code to be run by the prover and verifier. Using a formalization of λ• we prove that all well-typed λ• programs result in code that is secure under the standard cryptographic assumption of collision-resistant hash functions. We have implemented λ• as an extension to the OCaml compiler, and have used it to produce authenticated versions of many interesting data structures including binary search trees, red-black+ trees, skip lists, and more. Performance experiments show that our approach is efficient, giving up little compared to the hand-optimized data structures developed previously.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

Such a scenario can be supported using *authenticated data structures* (ADS) [5, 24, 31]. ADS computations involve two roles, the *prover* and the *verifier*. The mirror plays the role of the prover, storing the data of interest and answering queries about it. The client plays the role of the verifier, posing queries to the prover and verifying that the returned results are authentic. At any point in time, the verifier holds only a short *digest* that can be viewed as summarizing the current contents of the data; an authentic copy of the digest is provided by the data owner. When the verifier sends the prover a query, the prover computes the result and returns it along with a *proof* that the returned result is correct; both the proof and the time to produce it are linear in the time to compute the query result. The verifier can attempt to verify the proof (in time linear in the size of the proof) using its current digest, and will accept the returned result only if the proof verifies. If the verifier is also the data provider, the verifier may also update its data stored at the prover; in this case, the result is an updated digest and the proof shows that this updated digest was computed correctly. ADS computations have two properties. *Correctness* implies that when both parties execute the protocol correctly, the proofs given by the prover verify correctly and the verifier always receives the correct result. *Security*[1] implies that a computationally bounded, malicious prover cannot fool the verifier into accepting an incorrect result.

Authenticated data structures can be traced back to Merkle [18]; the well-known *Merkle hash tree* can be viewed as providing an authenticated version of a bounded-length array. More recently, authenticated versions of data structures as diverse as sets [23, 27], dictionaries [1, 12], range trees [16], graphs [13], skip lists [11, 12], B-trees [21], hash trees [26], and more [15] have been proposed. In

```
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

1. Maintaining a custom compiler frontend imposes development burden.
2. To construct compact proofs, the compiler implements several optimizations that are not covered by the security and correctness theorems.
3. Even with optimizations, the generated data structures are not always producing proofs as compact as hand-written implementations.

About Blog Publications

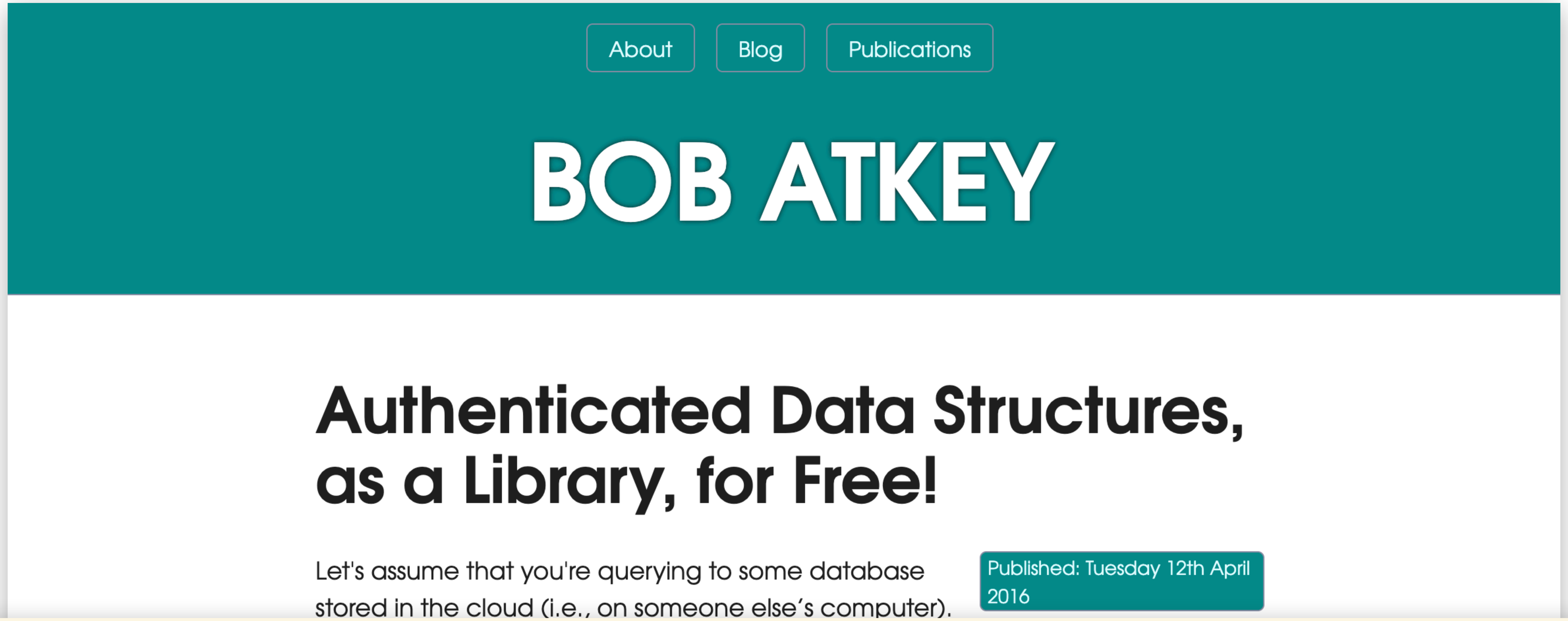
BOB ATKEY

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 type MERKLE = functor (A : AUTHENTIKIT) -> sig
  open A

  (* ... *)

  val fetch : path -> tree auth -> string option auth_computation
end
```

This work

- Two **logical-relations models** and a proof of security and correctness of the typed module construction in a general-purpose programming language.
- We address the remaining two limitations:
  - We verify several **optimizations** (as supported by the compiler).
  - We show how to prove that manually verified code can be **safely linked** with automatically generated code.
- Full mechanization in the Rocq theorem prover.

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


  (* ... *)





  (* ... *)





  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





  (* ... *)






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

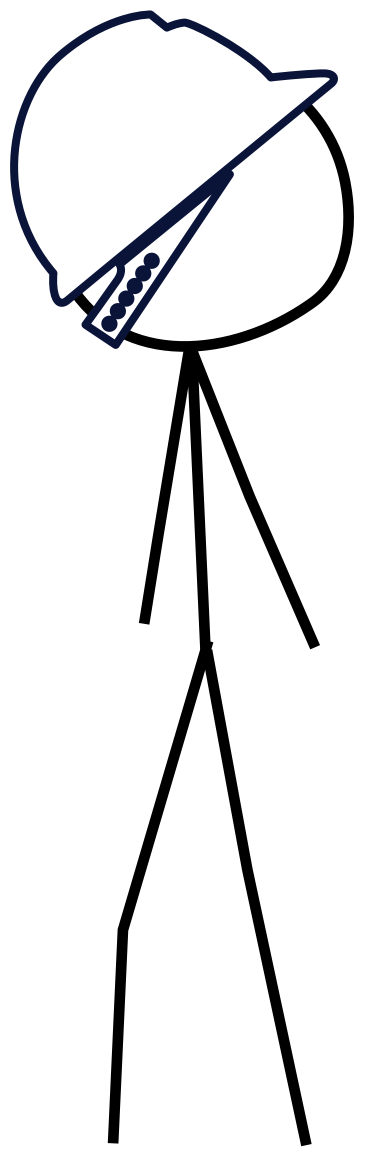
```
module type AUTHENTIKIT = sig
  type 'a auth
  type 'a auth_computation

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

  module Serializable : sig
    type 'a evidence
    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
```

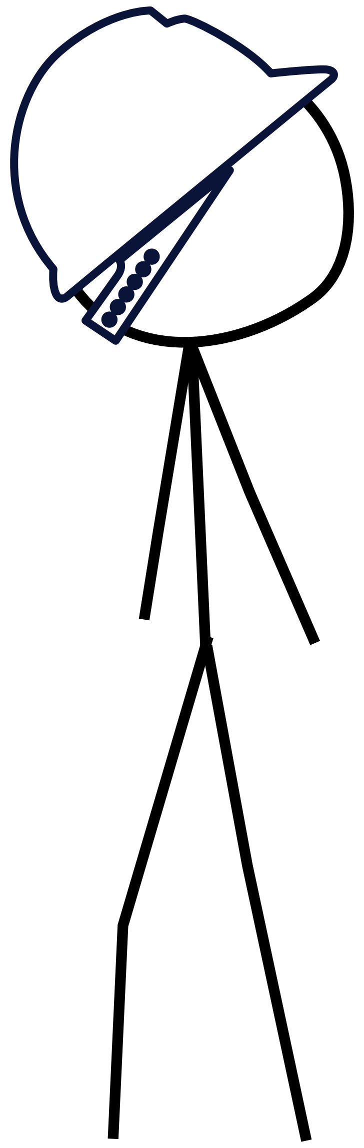
```
module Merkle : MERKLE = functor (A : AUTHENTIKIT) -> struct
  open A

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



   (* ... *)





   (* ... *)



end
```

```
module Merkle : 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.evi = Serializable.(sum string (pair auth auth))

  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
```

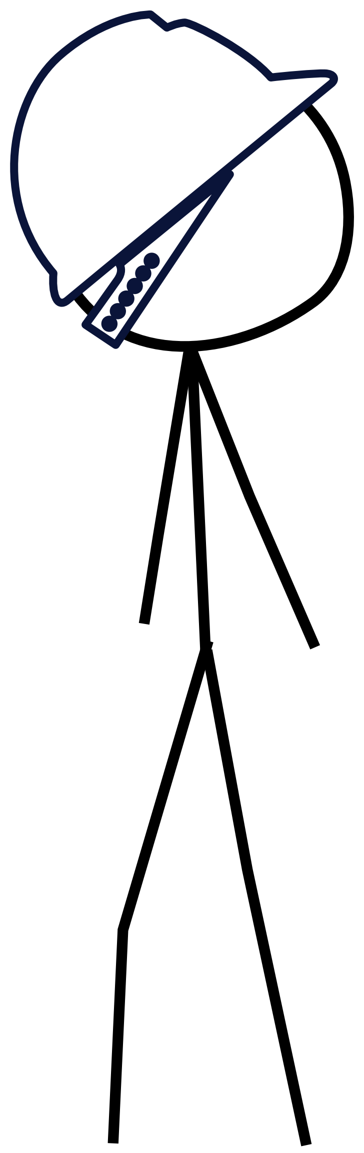
```
module Merkle : MERKLE = functor (A : AUTHENTIKIT) -> struct
  open A

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

  let tree_evi : tree Serializable.evi = Serializable.(sum string (pair auth auth))

  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
```

```
type proof = string list

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







  (* ... *)







  (* ... *)

end
```



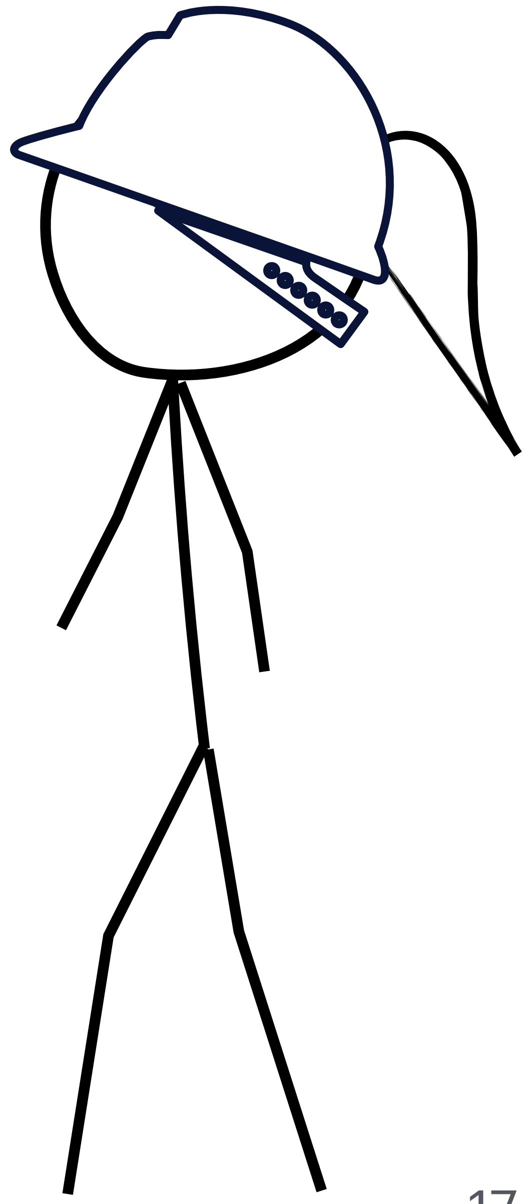


```
type proof = string list

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
```

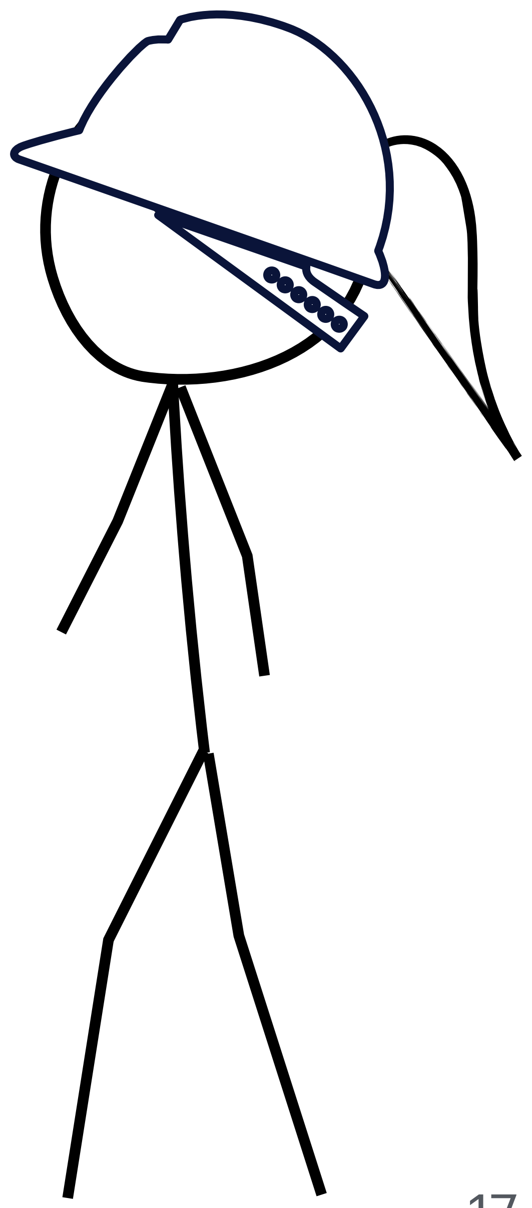


```
type proof = string list

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
```

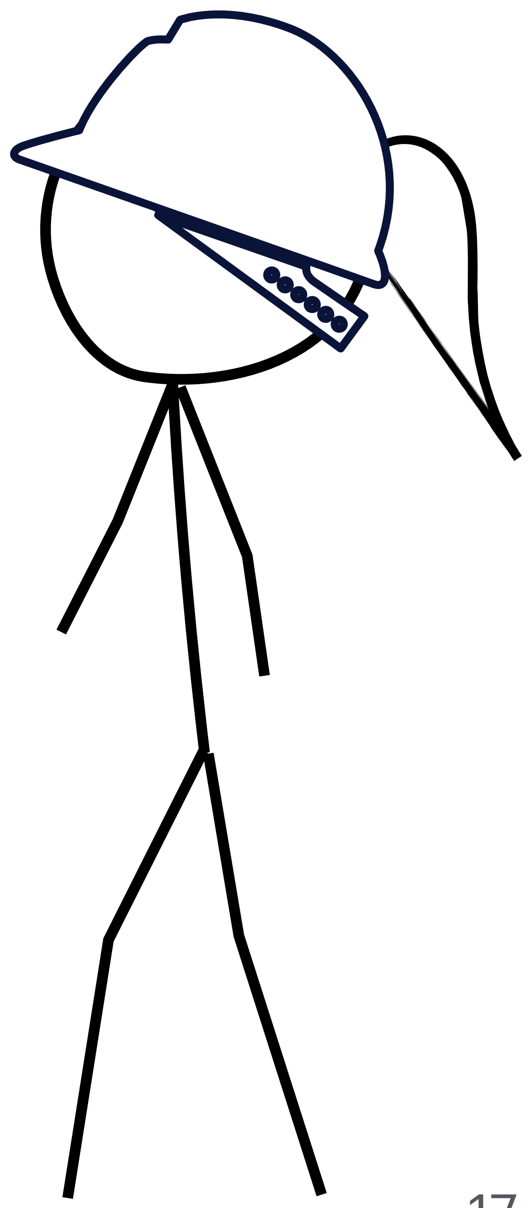
```
module Verifier : AUTHENTIKIT =
  type 'a auth = string
  type 'a auth_computation =
    proof -> [`Ok of proof * 'a | `ProofFailure]

  (* ... *)

  (* ... *)

end
```

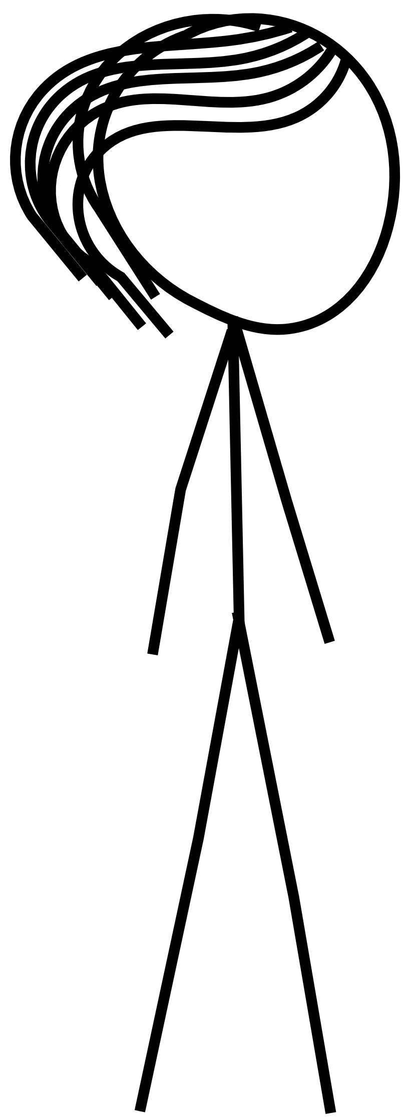


```
module Verifier : AUTHENTIKIT =
  type 'a auth = string
  type 'a auth_computation =
    proof -> [`Ok of proof * 'a | `ProofFailure]

  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

- 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.

Plan

1. Define a **type system** that can capture the module-based construction.

2. Define a **semantic model** that captures the type system.

3. Show that the inhabitants of the semantic model have the property of interest.

4. Show that the three Authentikit implementations inhabit the model.

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
    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
```

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
    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
```

(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
    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
```

polymorphism

(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
    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
```

abstract types

polymorphism

(higher-order) functions

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
    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
```

abstract types

recursive types

polymorphism

(higher-order) functions

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
```

(abstract) type constructors

abstract types

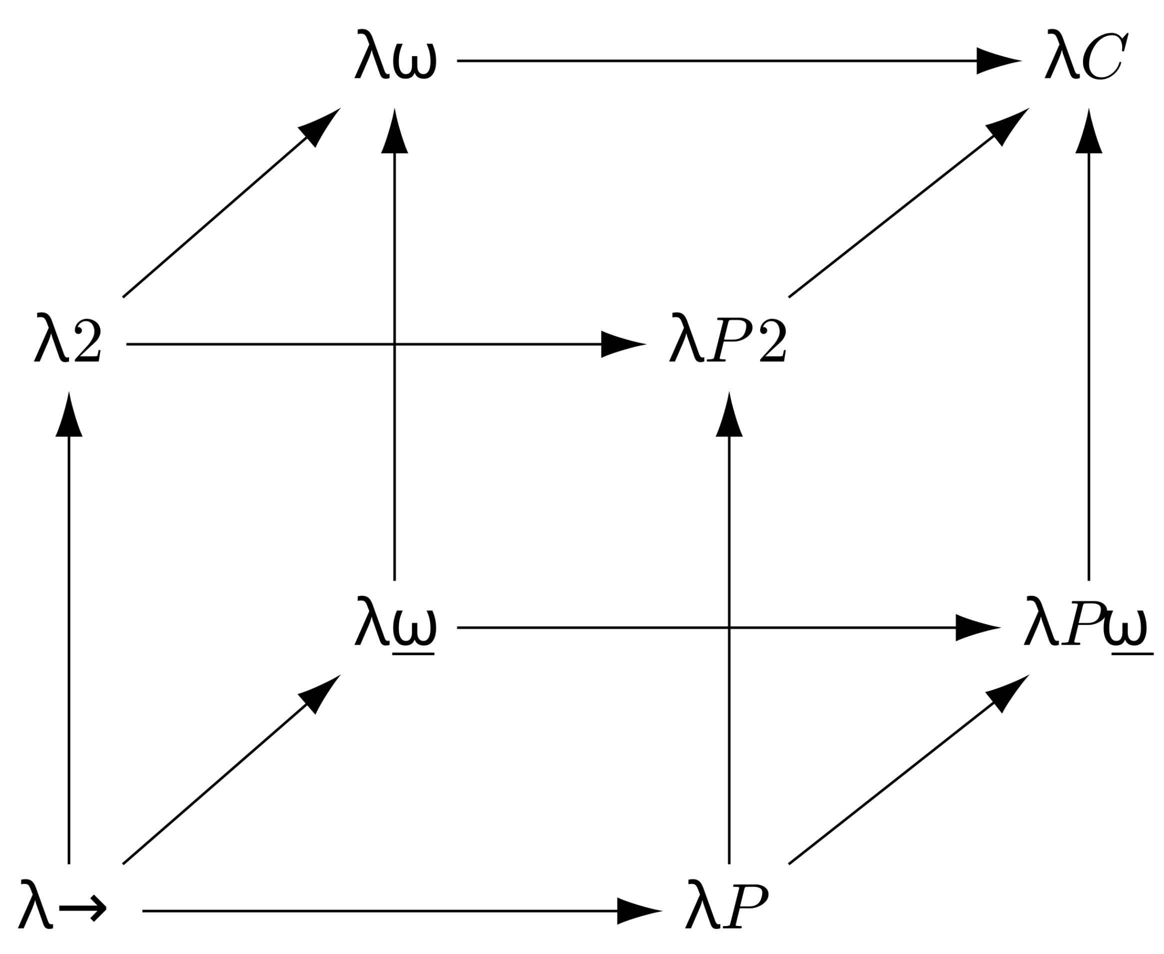
recursive types

```
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
    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
```

polymorphism

(higher-order) functions

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 $F_{\omega,\mu}^{\text{ref}}$ 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)

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

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

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

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

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

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

Our approach

To show security and correctness we

1. Define a **program logic** that is expressive enough for proving that programs have the property in question, e.g., a variant of Hoare logic.

2. Define a **semantic model** of the type system, in which types are given meaning through Hoare triples of the program logic.

Using the rules of the logic, we then show that the model is sound and that well-typed terms inhabit the model.

Collision-free reasoning

We first define a relational **Collision-Free Separation Logic (CF-SL)** on top of Iris.

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

CF-SL statements hold **"up to"** hash collision: given P holds for the initial state,

if e_1 evaluates to v_1 and e_2 evaluates to v_2 then $Q(v_1, v_2)$ holds

or a hash collision occurred.

Collision-f

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

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

We first define a relational **Collision-Free Separation Logic (CF-SL)** on top of Iris.

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

CF-SL statements hold **"up to"** hash collision: given P holds for the initial state,

if e_1 evaluates to v_1 and e_2 evaluates to v_2 then $Q(v_1, v_2)$ holds

or a hash collision occurred.

CF-SL

CF-SL satisfies all the standard program-logic rules, e.g.,

$$\frac{\{P\}\ e_1 \sim e_2'\ \{Q\} \qquad e_2 \rightsquigarrow e_2'}{\{P\}\ e_1 \sim e_2\ \{Q\}} \qquad\qquad \frac{\{\ell \mapsto w\}\ ()\sim e_2\ \{Q\}}{\{\ell \mapsto v\}\ \ell := w \sim e_2\ \{Q\}}$$

CF-SL

CF-SL satisfies all the standard program-logic rules, e.g.,

$$\frac{\{P\}\ e_1 \sim e_2'\ \{Q\} \qquad e_2 \rightsquigarrow e_2'}{\{P\}\ e_1 \sim e_2\ \{Q\}} \qquad \frac{\{\ell \mapsto w\}\ () \sim e_2\ \{Q\}}{\{\ell \mapsto v\}\ \ell := w \sim e_2\ \{Q\}}$$

but introduces a new proposition $\mathsf{hashed}(s)$ satisfying

$$\frac{\{P \star \mathsf{hashed}(s)\}\ \mathit{hash}(s) \sim e_2\ \{Q\}}{\{P\}\ \mathsf{hash}\ s \sim e_2\ \{Q\}} \qquad \frac{\mathit{collision}(s_1, s_2)}{\mathsf{hashed}(s_1) \star \mathsf{hashed}(s_2) \vdash \mathsf{False}}$$

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

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.

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

Theorem (Security)

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

$$\varnothing \mid \varnothing \vdash e : \forall \text{auth}, \text{m} \,.\, \text{Authentikit auth m} \rightarrow \text{m}\ \tau$$

then for all proofs p, if

$$e\ \ \text{Authentikit}_V\ \ p \rightarrow^*_{\text{cf}} \text{Some}(v)$$

then

$$e\ \ \text{Authentikit}_I \rightarrow^* v$$

Theorem (Correctness)

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

$$\varnothing \mid \varnothing \vdash e : \forall \text{auth}, \text{m} \,.\, \text{Authentikit auth m} \rightarrow \text{m}\ \tau$$

then if

$$e\ \text{Authentikit}_P \rightarrow^*_{\text{cf}} (p, v)$$

then

$$e\ \text{Authentikit}_V\ p \rightarrow^* \text{Some}(v) \quad \text{and} \quad e\ \text{Authentikit}_I \rightarrow^* v$$

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
```

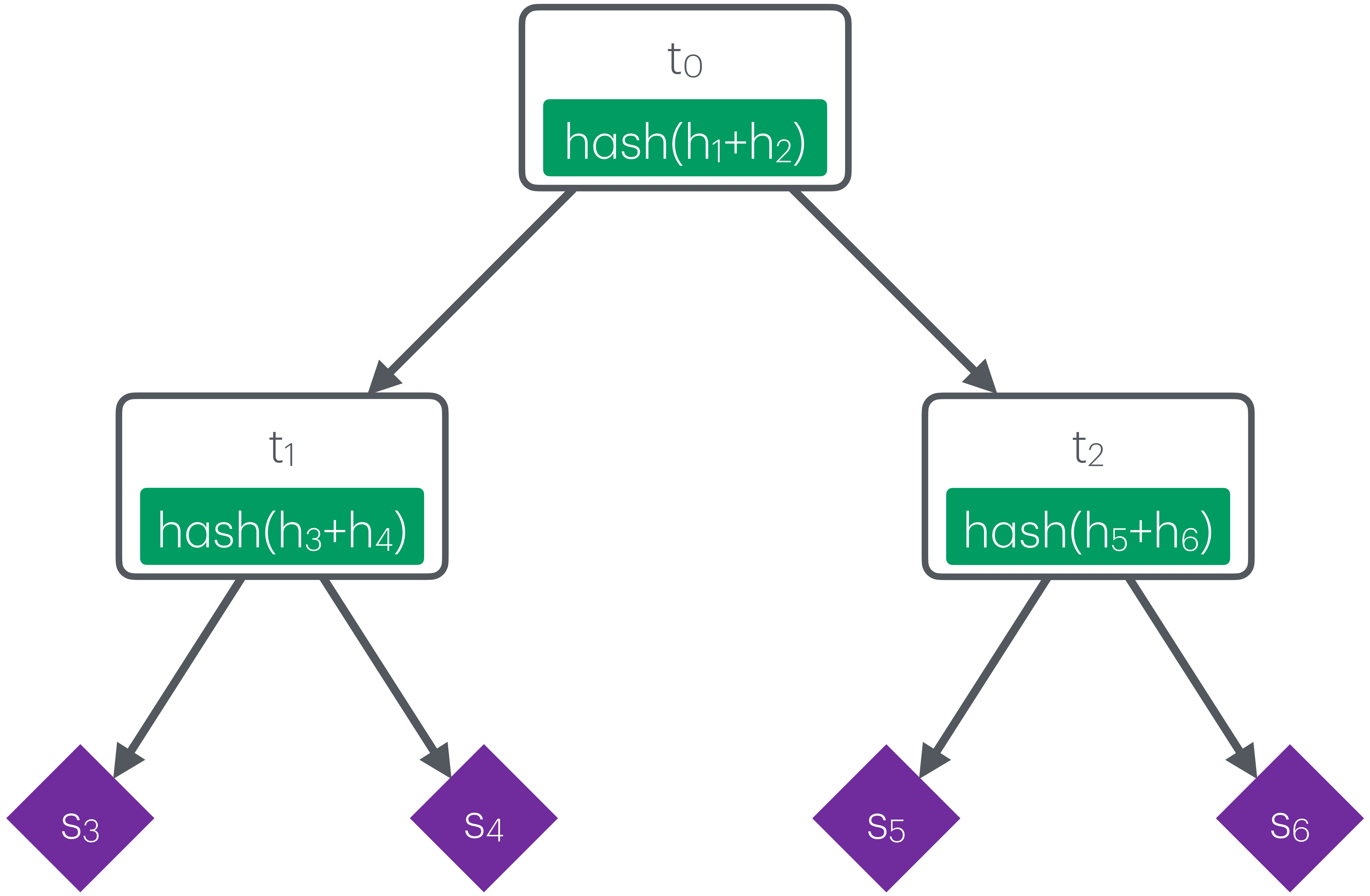
Manual proofs

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

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:

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



Summary

- **Authentikit** is a library for implementing ADSs generically.
- Two **logical-relations models** and a proof of security and correctness of the typed module construction in a general-purpose programming language.
  - We verify several **optimizations**.
  - We show how to prove that manually verified code can be **safely linked** with automatically generated code.
- Full mechanization in the Rocq theorem prover.


That's it, folks!