Authenticated Data Structures, Generically

Kyle Isom

The paper

Authenticated Data Structures, Generically
Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi
Published in 2014

CiteSeerX DOI: 10.1.1.394.2937

What is an ADS?

ADS: authenticated data structure

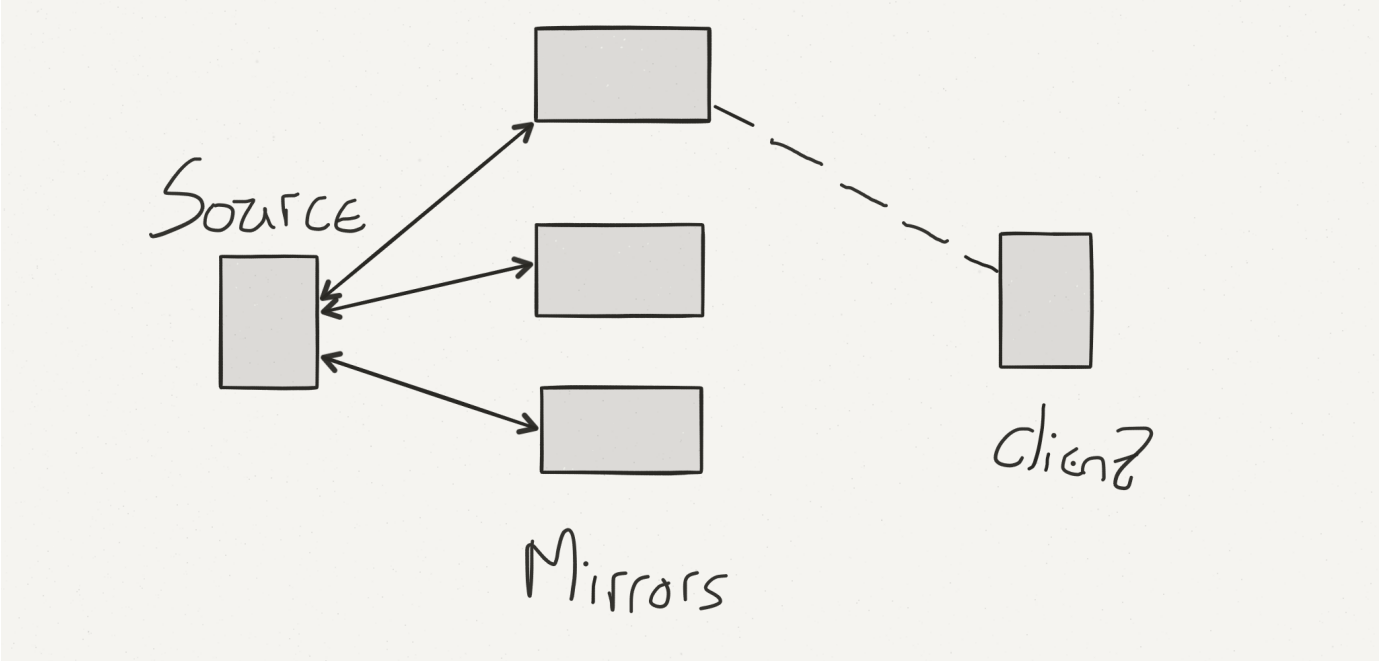
it's a **data structure**

- important that the data is useful to us

… that is **authenticated**

- A **prover** carries out some operation on the data
- A **verifier** can check the prover's work

Why use an ADS?



Why use an ADS?

Example: a PGP key server

- Source wants to scale for reliability (or DDoS protection)
- Client queries mirrors for key

Mirrors aren't necessarily trusted

- What if they inject their own key for someone?

Why use an ADS?

Client:

- Wants to ensure the integrity of keys being sent
- Wants to ensure that the keys came from the source

What have people done with them?

Built a wide variety of data structures

- sets
- dictionaries
- range queries
- B-trees

What have people done with them?

Most use cryptographic hash functions

- These are used for their collision resistance property
- Collision resistance: it is astronomically unlikely that two inputs to the hash function hash to the same value

What have people done with them?

There are some optimisations that can be done

- This paper doesn't cover most of them

λ●

Pronounced "lambda auth"

> "*a ML-like functional programming language... with which one can program authenticated operations over any data structure defined by standard type constructors*"

λ●

Implemented in the paper as an extension to the OCaml compiler.

Data structures written in λ● are compiled to code to be run by the prover and verifier.

Why implement ADSs via λ●?

- Rely on the static typing properties of OCaml
- A well-typed program generates correct and secure code for the prover and verifier
- Security is the same as for cryptographic hashes

Why implement ADSs via λ●?

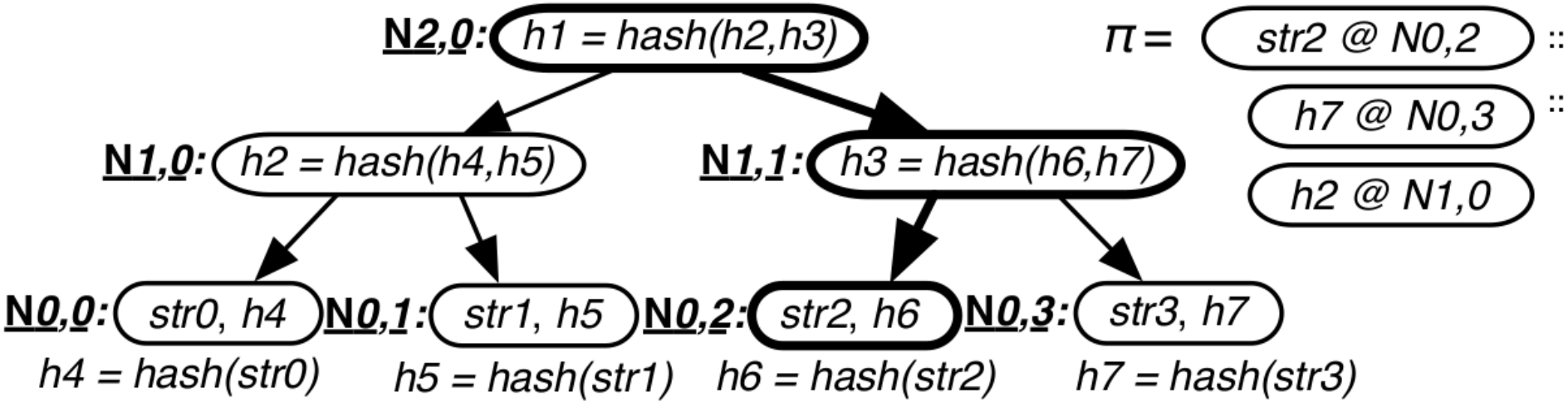
Authors propose two main benefits:

1. It is flexible; the authors implemented a number of data structures this way.
2. It is easy to use (if you're into the whole statically-typed functional programming thing)

Example ADS: Merkle Trees

- Merkle trees are authenticated binary trees
- Data is stored in leaves, not in nodes

Example ADS: Merkle Trees



Example ADS: Merkle Trees

The typical query looks up value at x_i

Example: $i = 1$

- Prover P returns x_1 and the digests π that are needed to prove the root digest
- Verifier computes the hashes
  - h_5 (from the leaf data)
  - the hash h_2 (by hashing h_4 and h_5)
  - the hash h_1 (by hashing h_2 and h_3)

Example ADS: Merkle Trees

The tree is balanced, so the size of the proof is $\log_2 n$, where n is the number of elements.

- Plugging in numbers: a SHA-256 hash is 32 bytes, so $\log_2(4) \rightarrow 2 * 32b = 64b$
- My PGP public key is 2215 bytes

Example ADS: Merkle Trees

In λ●:

```
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
```

How does λ● work?

λ● extends OCaml with *authenticated types* (●τ), and functions *auth* and *unauth*.

- *auth*: $\forall a.a \rightarrow \bullet a$
- *unauth*: $\forall a.\bullet a \rightarrow a$

How does λ● work?

On the prover, ●τ is stored as a (τ, hash) pair; `auth` and `unauth` are used to generate proofs.

On the verifier, ●τ is stored as a hash; `auth` and `unauth` are used to check a proof **π**.

Final thoughts

- ADS allow us to query data from an untrusted mirror so long as it originates from a trusted source

Haskell examples

Caveats:

- I haven’t written Haskell in a while
- AuthTypes is only an implementation of some core ideas; without compiler support, it’s difficult to do right