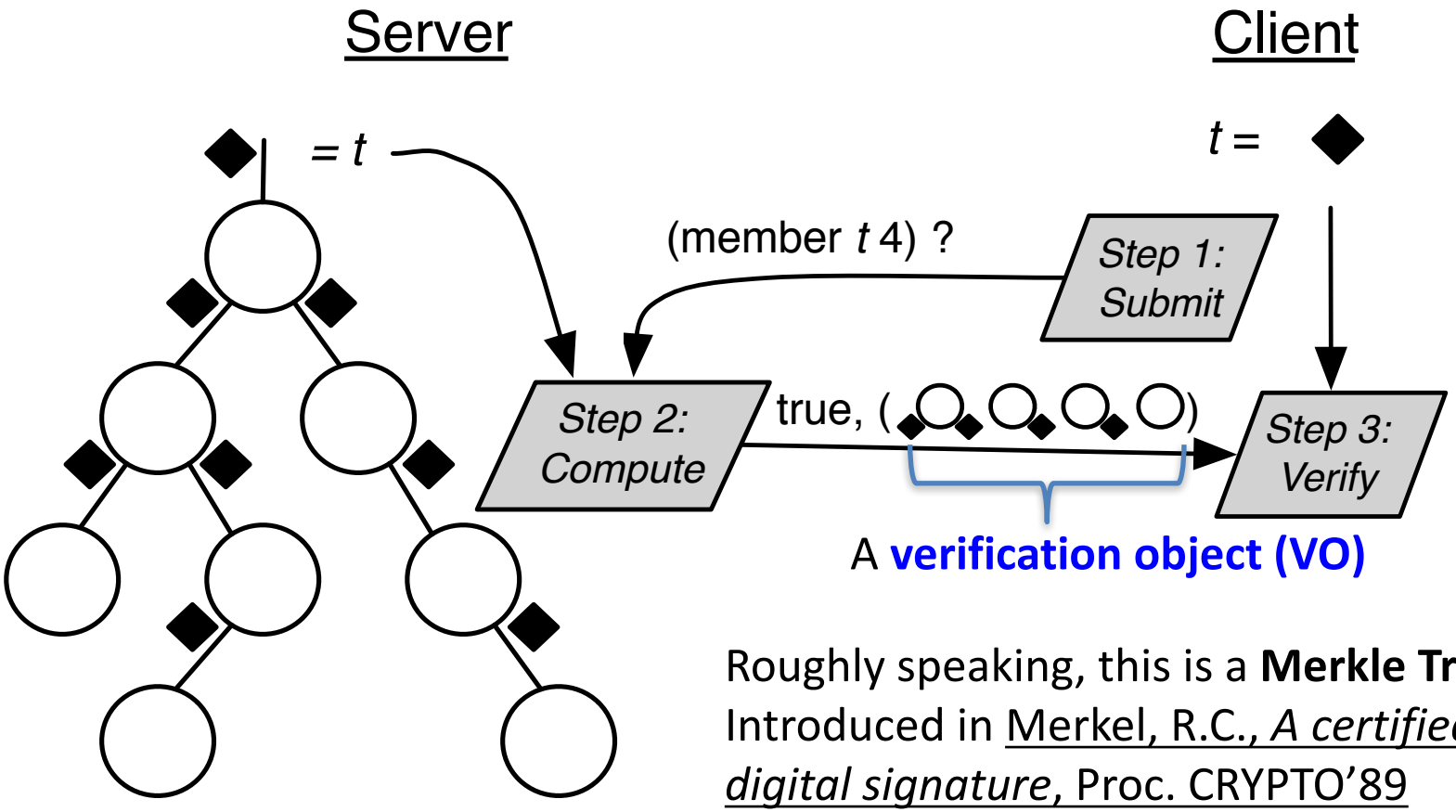
Authenticated Data Structures, Generically

Michael Hicks

with Andrew Miller, Elaine Shi, and Jonathan Katz

The University of Maryland, College Park

Authenticated Data Structures



Roughly speaking, this is a **Merkle Tree**
Introduced in Merkel, R.C., *A certified digital signature*, Proc. CRYPTO'89

Applications/benefits of ADSs

- Trustworthy mirroring/duplication
  - Duplicate data on many untrusted servers, but ensure trustworthy results (ensures integrity, not privacy)
  - Low space requirement for client
- Practical example: Bitcoin block chain
  - But implementation is suboptimal due to difficulty of combining the algorithm with the crypto
- Other applications possible: GPG keyservers, Tor relay directories, Tahoe-LAFS mutable files, …

Generic method for building ADSs?

- State of the art: design different ADSs in an ad hoc (and heroic) fashion
  - Litany of papers on improvements to existing data structures, or variations

- Instead: Can we add something to a programming language to make it easy to build new ADSs?
  - (Yes!)

Authenticated Data Structures, Generically. Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, January 2014. http://www.cs.umd.edu/~mwh/papers/miller14gpads.html

Approach

- Start with a pure functional programming language with standard features
  - E.g., Ocaml with datatypes, (recursive) functions, base types, etc. but no refs
- Add new type $\bullet\tau$ and two coercions
  - ***auth***: $\tau \longrightarrow \bullet\tau$
  - ***unauth***: $\bullet\tau \longrightarrow \tau$
- Develop two **computation modes**, one for the server, one for the client
  - Compiler produces code for each mode

Example: BST with auth types

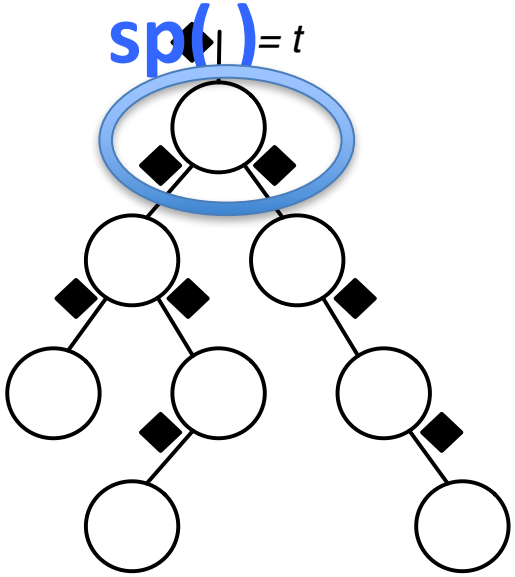
```
type tree =
    | Tip
    | Bin of •tree × int × •tree

let rec member (t:•tree) (x:int) : bool =
    match unauth t with
     Tip → false
    | Bin (l,y,r) →
        if y = x then true
        else if x < y then member l x
        else member r x

let rec insert (t:•tree) (x:int) : •tree =
    match unauth t with
     Tip → auth (Bin(auth Tip,x, auth Tip))
    | Bin (l,y,r) →
        if y = x then t
        else if x < y then auth (Bin(insert l x,y,r))
        else auth (Bin(l,y,insert r x))
```

Execution modes

Server (mode **P**rover) produces VO.

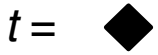
A value of type $\bullet\tau$ is

- a value v of type τ, coupled with
- a cryptographic hash *d* of its shallow projection *sp*(v)
  - Informally: serialize the data up to, but not past, nested authenticated values, and hash that



Client (mode **V**erifier) consumes VO.

A value of type $\bullet\tau$ is a hash *d*



Proof and Verification

- The VO is a list of shallow projections of authenticated values
- Server (mode P): unauth <d,v> enqueues *sp*(v) on the VO
  - Returns v
- Client (mode V): unauth d checks that hash(hd(VO)) = d
  - Dequeues and returns the head of the VO

Formalization

- Extension to standard programming language
  - auth and unauth, simple typing rules
- Small-step operational semantics
  - Three variants, indexed by modes I, P, V
  - Differ on auth, unauth, and whether *VO* consumed/produced
- **Theorem**: Starting with a well-typed program
  - The client/server together compute the right answer under normal operation
  - The client will only accept the wrong answer if the server can manufacture a hash collision (very hard!)

Security Proofs

- Proofs of key theorems use standard, syntactic methods common in programming languages
  - Like those we saw in earlier lectures
  - Taught in first graduate class on semantics
- Methods designed to speak about whole classes of computation, not single programs
  - Proof done once, for the language, not per ADS

Syntax and Types

$$
\begin{array}{rcl}
\text{Types } \tau & ::= & 1 \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mu\alpha.\tau \mid \alpha \mid \bullet\tau \\
\text{Values } v & ::= & () \mid x \mid \lambda x.e \mid \mathbf{rec}\ x.\lambda y.e \\
& \mid & \mathbf{inj}_1\ v \mid \mathbf{inj}_2\ v \mid (v_1, v_2) \mid \mathbf{roll}\ v \\
\text{Exprs } e & ::= & v \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid v_1\ v_2 \mid \mathbf{case}\ v\ v_0\ v_1 \\
& \mid & \mathbf{prj}_1\ v \mid \mathbf{prj}_2\ v \mid \mathbf{unroll}\ v \mid \mathit{auth}\ v \mid \mathit{unauth}\ v
\end{array}
$$

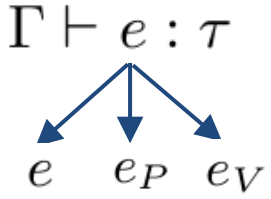
$$\frac{\Gamma \vdash v : \tau_1}{\Gamma \vdash \mathbf{inj}_1\ v : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash v : \tau_2}{\Gamma \vdash \mathbf{inj}_2\ v : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash v : \tau_1 + \tau_2 \qquad \Gamma \vdash v_1 : \tau_1 \to \tau \qquad \Gamma \vdash v_2 : \tau_2 \to \tau}{\Gamma \vdash \mathbf{case}\ v\ v_1\ v_2 : \tau}$$

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathit{auth}\ v : \bullet\tau} \qquad \frac{\Gamma \vdash v : \bullet\tau}{\Gamma \vdash \mathit{unauth}\ v : \tau}$$

...

Operational Semantics in 3 Modes



Input program

"Compilation"

$$\ll \pi, \mathbf{case}\ (\mathbf{inj}_1\ v)(\lambda x.e_1)(\lambda x.e_2) \gg \ \rightarrow_m \ \ll \pi, e_1[v\backslash x] \gg$$
$$\ll \pi, \mathbf{case}\ (\mathbf{inj}_2\ v)(\lambda x.e_1)(\lambda x.e_2) \gg \ \rightarrow_m \ \ll \pi, e_2[v\backslash x] \gg$$
$$\ll \pi, \mathbf{prj}_1\ (v_1, v_2) \gg \ \rightarrow_m \ \ll \pi, v_1 \gg$$
$$\ll \pi, \mathbf{prj}_2\ (v_1, v_2) \gg \ \rightarrow_m \ \ll \pi, v_2 \gg$$

- m is either I, P, or V
- carry around the VO π
- most transitions leave it unchanged

$$\ll \pi, auth\ v \gg \ \rightarrow_I \ \ll \pi, v \gg$$
$$\ll \pi, unauth\ v \gg \ \rightarrow_I \ \ll \pi, v \gg$$

auth/unauth are no-ops in ideal mode

Operational Semantics in 3 Modes

$$
\begin{array}{lcl}
(\!|()|\!) & = & () \\
(\!|\langle h, v\rangle|\!) & = & h \\
(\!|auth\ v|\!) & = & auth\ (\!|v|\!) \\
(\!|(v_1, v_2)|\!) & = & ((\!|v_1|\!), (\!|v_2|\!)) \\
(\!|\mathbf{roll}\ v|\!) & = & \mathbf{roll}\ (\!|v|\!) \\
(\!|\mathbf{rec}\ x.\lambda y.e|\!) & = & \mathbf{rec}\ x.(\!|\lambda y.e|\!)
\end{array}
$$

...

Shallow projection function, written (|e|)

$$
\begin{array}{lll}
\ll \pi, auth\ v \gg & \rightarrow_P & \ll \pi, \langle \mathsf{hash}\ (\!|v|\!), v\rangle \gg \\
\ll \pi, auth\ v \gg & \rightarrow_V & \ll \pi, \mathsf{hash}\ v \gg
\end{array}
$$

auth in Prover/Verifier builds new digest

$$
\ll \pi, unauth\ \langle h, v\rangle \gg \quad \rightarrow_P \quad \ll \pi\ @\ [\ (\!|v|\!)\ ],\ v \gg
$$

unauth in Prover adds to the stream

$$
\frac{\mathsf{hash}\ s_0 = h}{\ll [s_0]\ @\ \pi, unauth\ h \gg \rightarrow_V \ll \pi, s_0 \gg}
$$

unauth in Verifier consumes from the stream

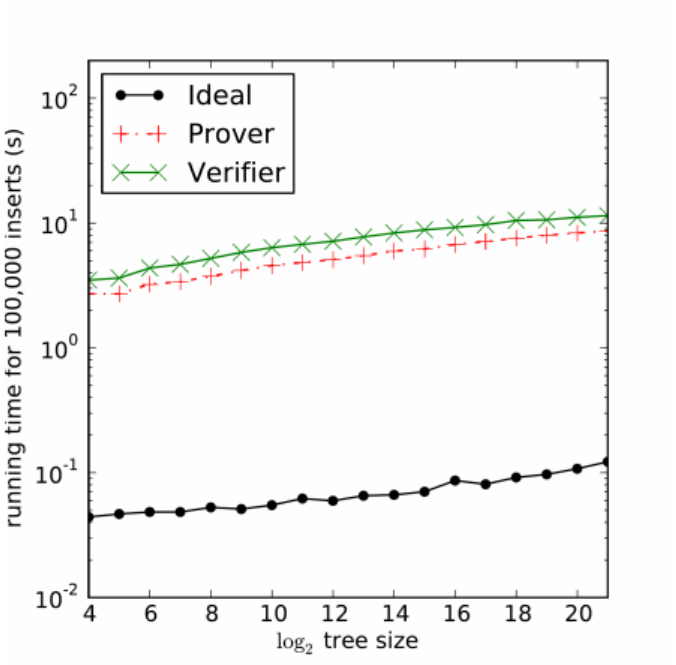
Implementation

- We have extended the Ocaml compiler to support authenticated types
  - produces versions of data structure for the *prover* (server) and *verifier* (client)
- Implemented several ADSs
  - BST, Red-black trees, skip lists, planar separator DS for shortest paths, efficient blockchain
- Confirmed expected space/time performance

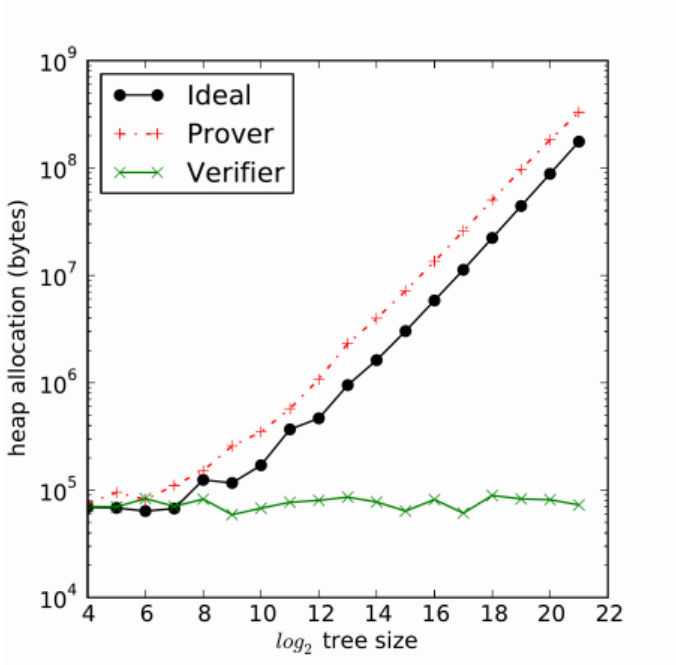
Red+black Tree Performance

Verifier:
- 55% in SHA1
  - used in both *auth* and *unauth*
- 30% in Marshal (used in hash(v))

Prover:
- 28% in SHA1
  - used in *auth* only
- 30% in Marshal
- 22% in GC



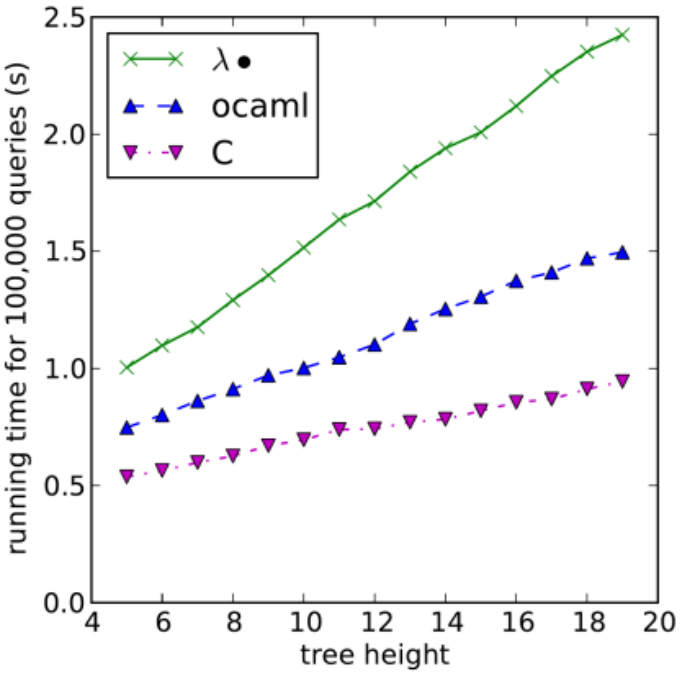
(a) Running time



(b) Memory usage

Merkle Tree Performance

Compared against
hand-rolled
C and OCaml



Summary

- Generic implementation of hash-based ADS
- Write program once in ordinary functional language, automatically derive Client/Server modes
- Once-and-for-all Security Theorem applies to every program
- Performance comparable to hand-optimized