Authenticated Data Structures, Generically

Andrew Miller

with Michael Hicks, Elaine Shi, and Jonathan Katz

The University of Maryland, College Park

POPL 2014

Motivation - What's an ADS?

Queries

Responses

Untrusted Server
Plenty of storage

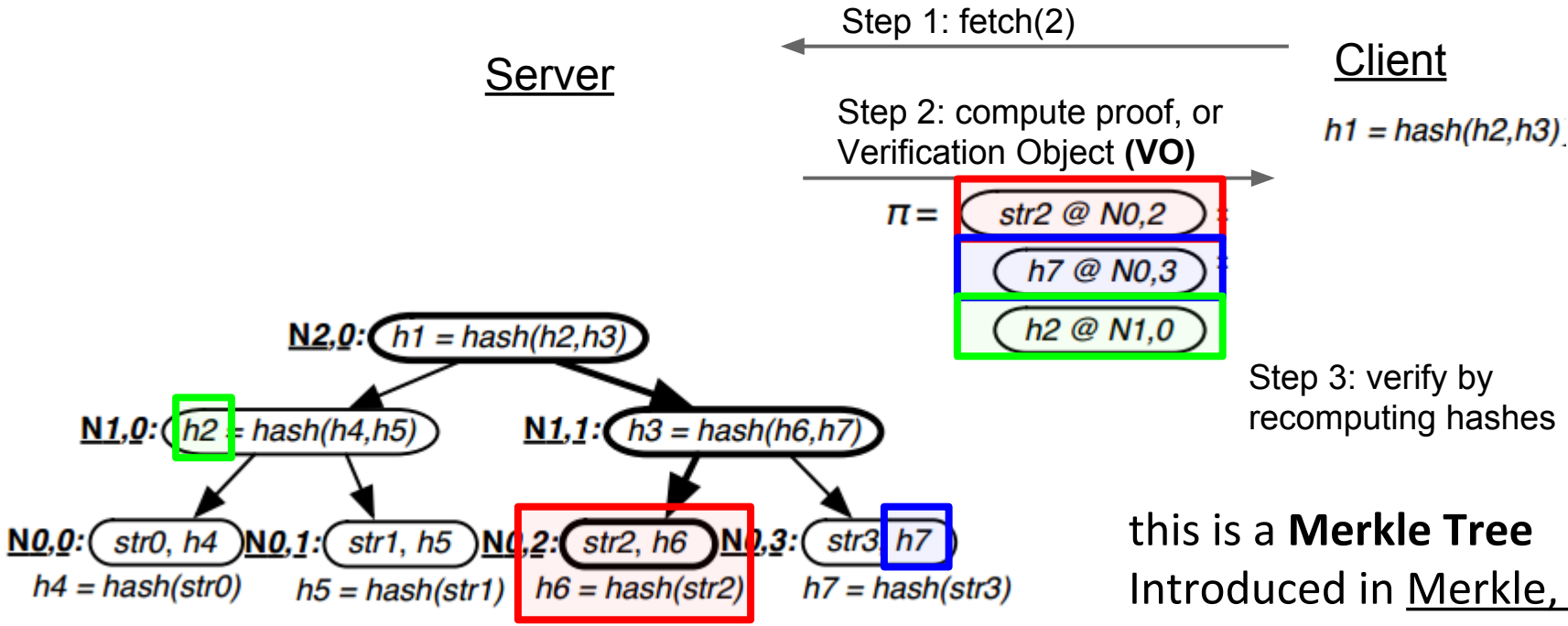
Trusted Client
Not much storage

Goal: trustworthy outsourced data storage

Bad approach: fetch the entire dataset, check a hash

Smart approach: only fetch data relevant to the query

An Authenticated Data Structure



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

Applications/benefits of ADS

- Trustworthy mirroring/duplication
  - Duplicate data on many untrusted servers, but ensure trustworthy results (ensures integrity, not privacy)
  - Low space requirement for client
- Examples: Tahoe-LAFS, BitTorrent, Amazon Dynamo, Bitcoin block chain (but a sub-optimal implementation!)
- Others possible: GPG keyservers, Tor relay directories, …

Generic method for building ADS?

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

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

Presenting λ• ("lambda auth")

- Purely functional, ML-like language
  - –Small extension for ADS support
- Compiler produces versions of data structure for the *prover* (server) and *verifier* (client)
  - –Formalized semantics as different evaluation modes
  - –Well-typed programs are correct and secure
- Implementation for Ocaml (preprocessor)
  - –Coded up new and existing ADS (easily, in most cases)

Example: Binary tree 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
```

- Start with a pure functional language
  - E.g., Ocaml with datatypes, (recursive) functions, base types, etc. but no refs
- Add new type $\bullet\tau$ ("auth τ"), coercions
  - *auth*: $\tau \longrightarrow \bullet\tau$
  - *unauth*: $\bullet\tau \longrightarrow \tau$
- Evaluation mode for prover, verifier.
  - Prover produces VO, verifier consumes/checks it
  - Result should relate to "ideal" mode

Ideal mode: Data and operations

- Authenticated types are the identity

type $\bullet\tau = \tau$

auth x = x

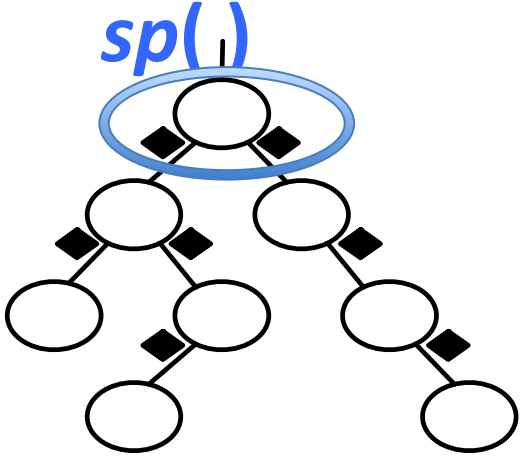
unauth x = x

- As such, easy to reason about what ADS is doing

Prover and Verifier: Data

At Prover, a value of type •τ is <d,v> where

- value v has type τ
- d is a *cryptographic hash* of v's shallow projection, written ***sp***(v)
  - Informally: serialize the data up to, but not past, nested authenticated values, and hash that
  - Pictorially sometimes write ♦ for d



At Verifier, a value of type •τ is a hash d

t = ◆

auth

- Prover: auth v returns <d,v> where d = *hash(sp*(v))
- Verifier: auth v returns d where d = *hash(*v)

unauth

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

PL for Crypto people… λ

λ: a simple (turing equiv.) computing model

Terms := λx.e | e e’ | x

Reduction: $(\lambda x.e)e' \longrightarrow e[e'\backslash x]$

Efficient Church-Turing thesis:
Polynomial number of reductions in polynomial turing machine steps.

An Invariant Cost Model for the Lambda Calculus Dal Lago and Martini, 2006, Second Conference on Computability in Europe.

PL for Crypto people… Types

Types: syntactic classes of programs

$$\text{Types } \tau \quad ::= \quad 1 \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mu\alpha.\tau \mid \alpha \mid \bullet\tau$$

Rules for program composition:

$$\frac{\Gamma \vdash v : \tau_1 \to \tau_2 \qquad \Gamma \vdash v' : \tau_1}{\Gamma \vdash v\ v' : \tau_2} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

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

and so on….

Type soundness:
Reduction preserves types

PL for Crypto people… editorial

Why another computing model?

1. Better fits the computational model
    - Typed λ is closer to OCaml than RAM is to C
    - This means we can use types in our formal theory
    - Gain a performance benefit vs naive translation (Later!)
2. Functional program language → correct implementations
    - Increasing popularity: F#, Scala, Map-Reduce
    - Already widely used in financial industry
    - Stepping stone to formal verification of implementations

Formalization

- Small extension to CBV, simply-typed lambda calculus with standard type constructors
  - –A-normal form for simplicity
- Operational semantics
  - –Three variants, indexed by modes I, P, V
  - –VO as side effect $\ll \pi, e \gg \rightarrow_m \ll \pi', e' \gg$
- Proof of correctness, security

Syntax and Types

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

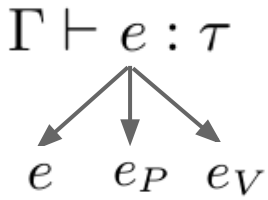
$$\frac{\Gamma \vdash v : \tau_1}{\Gamma \vdash \textbf{inj}_1\ v : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash v : \tau_2}{\Gamma \vdash \textbf{inj}_2\ v : \tau_1 + \tau_2}$$

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

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

...

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

3-Way Agreement Relation

$$\Gamma \vdash () \; () \; () : 1$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \; x \; x : \tau}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \;\; e_P \;\; e_V : \tau_2}{\Gamma \vdash (\lambda x.e) \; (\lambda x.e_P) \; (\lambda x.e_V) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash v \;\; v_P \;\; v_V : \tau_1}{\Gamma \vdash (\mathbf{inj}_1 \; v) \; (\mathbf{inj}_1 \; v_P) \; (\mathbf{inj}_1 \; v_V) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash v \;\; v_P \;\; v_V : \tau_1 \times \tau_2}{\Gamma \vdash (\mathbf{prj}_1 v) \; (\mathbf{prj}_1 \; v_P) \; (\mathbf{prj}_1 \; v_V) : \tau_1}$$

...

most terms preserve self-agreement

$$\frac{\vdash v \;\; v_P \;\; (\!|v_P|\!) : \tau \qquad \mathsf{hash}\,(\!|v_P|\!) = h}{\Gamma \vdash v \;\; \langle h, v_P \rangle \;\; h : \bullet\tau}$$

Special case for auth-type agreement

Security Theorem (informal)

- Correctness: if Prover runs correctly, then Verifier gets the correct answer
  - Prover's and Verifier's final values *agree* with Ideal

- Security: if the Verifier gets an incorrect answer, then we can extract a hash collision
  - Computationally hard to do: implies security

Security Theorem

Suppose we start with in-agreement programs $\vdash e\ e_P\ e_V : \tau$

Correctness: If in Ideal mode $\ll [], e \gg \rightarrow_I^i \ll [], e' \gg$,

$\ll [], e_P \gg \rightarrow_P^i \ll \pi, e'_P \gg$

$\ll \pi, e_V \gg \rightarrow_V^i \ll [], e'_V \gg$

$\vdash e'\ e'_P\ e'_V : \tau$

then we can run Prover, and give its output to Verifier, get correct answer.

Security: If for for a possibly malicious prover, $\ll \pi_{\mathcal{A}}, e_V \gg \rightarrow_V^i \ll \pi', e'_V \gg$

then either:

(verifier is correct)

$\ll [], e \gg \rightarrow_I^i \ll [], e' \gg$

$\ll [], e_P \gg \rightarrow_P^i \ll [] \mathbin{@} \pi, e'_P \gg$

$\pi_{\mathcal{A}} = \pi \mathbin{@} \pi'$

$\vdash e'\ e'_P\ e'_V : \tau$

or:

(we can find a hash collision)

$j \leq i,$

$\ll [], e_P \gg \rightarrow_P^j \ll [] \mathbin{@} \pi_0 \mathbin{@} [s], e'_P \gg$

$\pi_{\mathcal{A}} = \pi_0 \mathbin{@} [s^\dagger] \mathbin{@} \pi'$

$s \neq s^\dagger$ *but* hash s = hash $s^\dagger$.

Implementation

- We have extended the OCaml compiler to support authenticated types
  - Do not handle authenticated closures, or polymorphism, but could (eventually)
- Implemented several ADSs
  - BST, Red-black trees, skip lists
  - Building planar separator DS for shortest paths (Novel ADS!)
- Confirmed expected space/time performance

Implementation

```
type •α = | Digest of string (* the digest *)
          | Prover of string × α

let auth_prover (shallow: α → α) (v:α) : •α =
  Prover(hash (shallow v), v)

let unauth_prover (shallow: α → α) (v:•α) : α =
  let Prover(_,x) = v in
  to_channel !prf_output (shallow x);
  x

let auth_verifier (v:α) : •α = Digest(hash v)

let unauth_verifier (v:•α) : α =
  let Digest(h) = v in
  let y = from_channel !prf_input in
  assert h = hash y;
  y

let shallow_• (Prover(h,_): •α) : •α = Digest(h)
```

```
(* User-provided code *)
type bst = Tip
  | Bin of •bst × int × •bst
  | AuthBin of •(bst × int × bst)
let is_empty (t:•bst) : bool = (unauth t = Tip)
let mk_leaf (x:int) : •bst = AuthBin(auth(Tip, x, Tip))

(* Generated Prover code *)
let rec shallow_bst : bst → bst = function
  | Tip → Tip
  | Bin (x, y, z) → Bin(shallow_• x, y, shallow_• z)
  | AuthBin (x) → AuthBin (shallow_bst1 x)
and shallow_bst1 : bst × int × bst → bst × int × bst
= function (x, y, z) → (shallow_bst x, y, shallow_bst z)

let unauth_bst = unauth_prover shallow_bst
let auth_bst1 = auth_prover shallow_bst1
```

Examples

- **Binary search tree (including updates)**
- Randomized skiplist (assume same seed)
- Not just trees: merge two lists
- Not just trees: sharing
- Novel ADS: planar separator tree
- Practical impact: Bitcoin

Example: BST insert

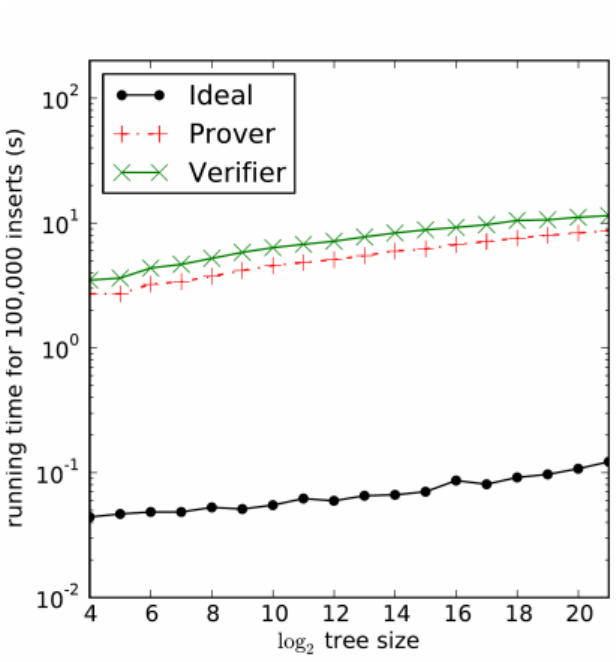
```
type tree =
   | Tip
   | Bin of •tree × int × •tree

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))
```

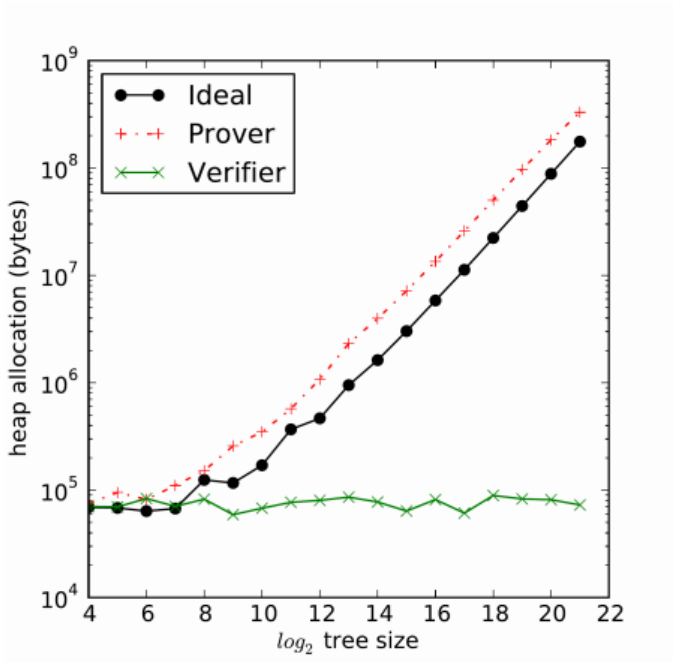
RedBlack+ tree performance

Verifier:
- 55% in SHA1
  - used in both *auth* and *unauth*
- 30% in Marshal
- Tags?

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



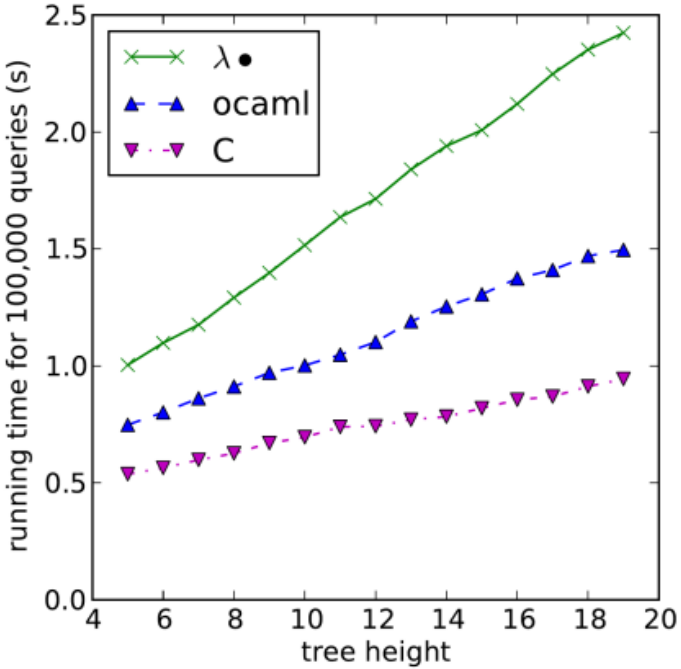
(a) Running time



(b) Memory usage

Merkle trees performance

Compared against
hand-rolled
C and OCaml



Better than naive translation

Standard approach:

RAM as computing model

Build a Merkle tree over RAM - O(log n) per access

Consider binary search on sorted data - $O(\log^2 n)$

Our approach:

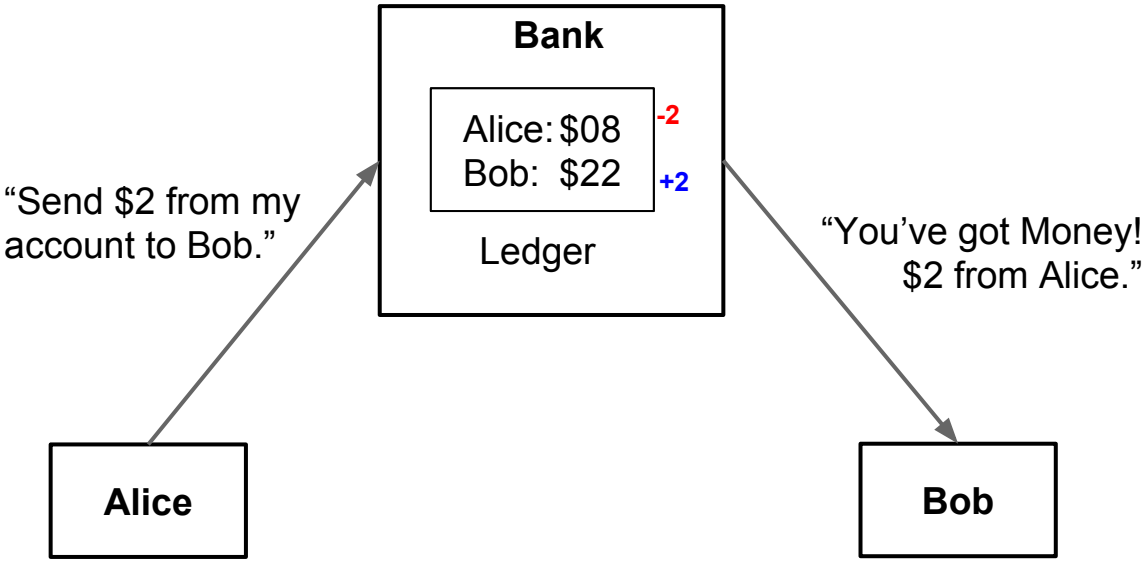
Merkle tree is interwoven with functional data structure

Only O(log n)

Part II: Bitcoin in 5 steps

(joint work with Narayanan, Kroll, Felten, Bonneau at CITP Princeton, and Clark at Concordia)

Ideal Bank Account Functionality

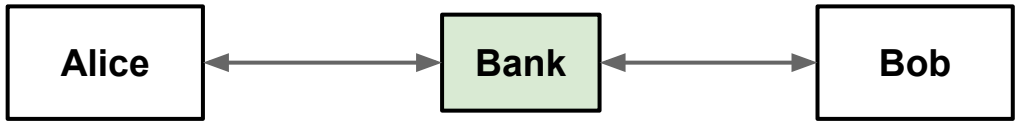


A single transaction: Ledger -> Ledger’ (or failure)
Valid transactions don’t spend more money than they have.

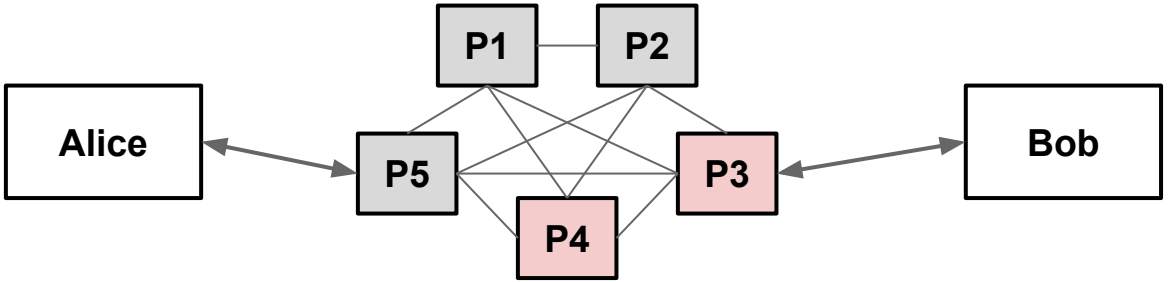
From Ideal to Bitcoin in 5 Steps

1. Implement the Bank as a trusted third party

(e.g., Paypal)



2. Implement the Bank as a multiparty computation

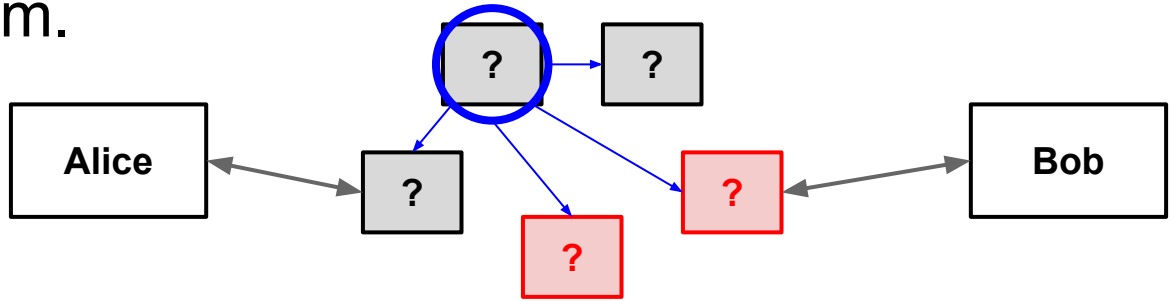


- Standard results in Byzantine fault-tolerance apply here, (e.g. ***Paxos***)

- PKI is assumed

From Ideal to Bitcoin in 5 Steps

3. Suppose we have a magic token that chooses parties at random.



*caveats

Whoever has the token gets to broadcast *once*

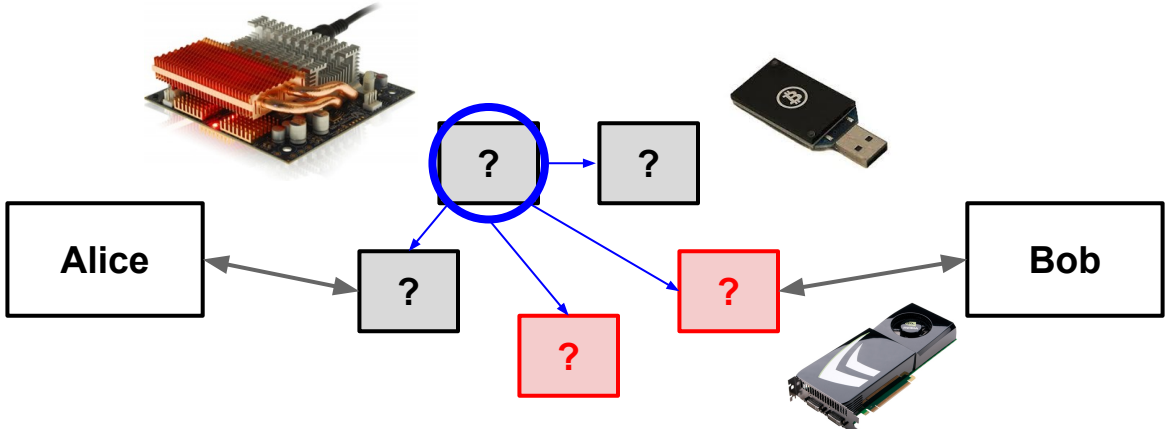
If t parties are malicious, Pr[honest selected] = (n-t)/t

Thm. *If majority are honest, transaction log converges*

From Ideal to Bitcoin in 5 Steps

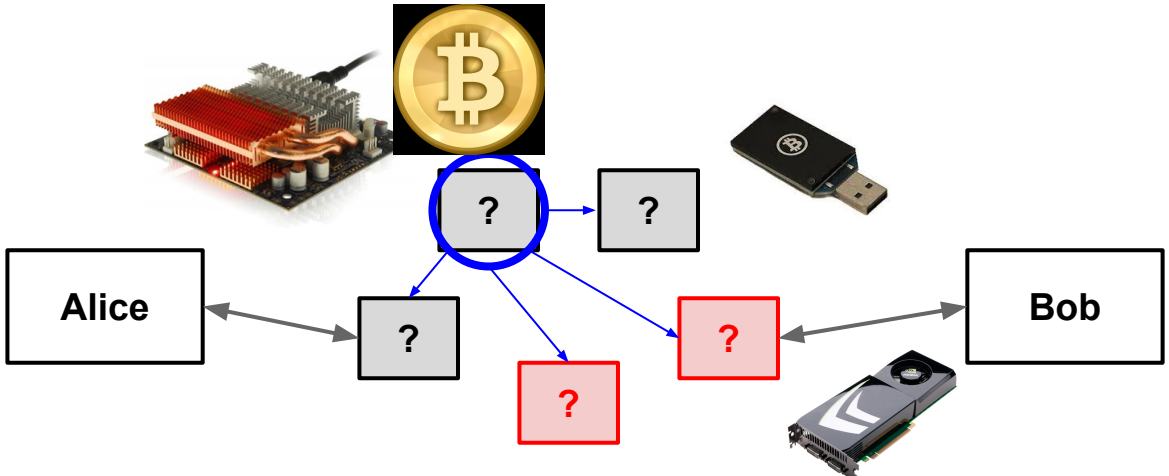
4. Replace the token with computational Scratch-off Puzzle
   - Solvable by concurrent/independent participants
   - No advantage over brute force

$\textbf{Scratch}_{d}(puz, m)$: $r \leftarrow \{0,1\}^{k}$; if $H(puz \,||\, m \,||\, r) < 2^{k-d}$ then return r



From Ideal to Bitcoin in 5 Steps

5. Finally, provide participation incentives
   - give each “lottery winner” a reward
   - also solves the problem of initial allocation
   - Incentive compatible participation?



Slightly More Detail

Ledger: state file, mapping amounts of BTC to pkeys
Transactions: Signed instructions to modify the ledger
Blockchain: Authenticated sequential log of transactions

Each solution is used as seed for the next puzzle challenge.
The solutions form linked lists (blockchains).

Thm. *For all n, eventually converge on unique n-length chain.*

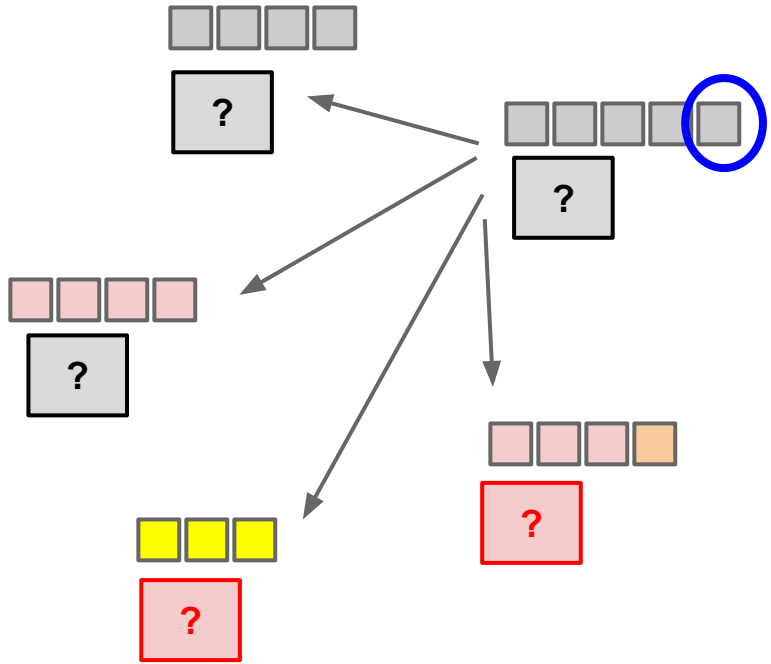
Consensus Protocol

Algorithm for process P_i

initially:
  preferred := { }
  txes := **input**

on **receive**(*chain*):
 parse *chain* as linked list
 if *chain* is **valid** and
 |*chain*| > |preferred| then
  preferred := *chain*

mainloop(): (as fast as possible)
 puz := H(preferred)
 r ← **Scratch**(puz, txes)
 if r ≠ ⊥ then
  preferred := preferred U {(txes, r)}
  broadcast preferred



Summary so far

- Bulk of existing fault tolerant distributed computing research (including malicious SMC) has focused on “eponymous” networks with PKI

- Anonymous networks are an open area

Exceptions:

- Okun. Agreement among unacquainted Byzantine generals. Distributed Computing, 2005.

- Aspnes et al. Exposing Computationally Challenged Byzantine Impostors. Yale TR, 2005.

- Delporte-Gallet et al. Byzantine agreement with homonyms. PODC 2011.

Is it actually incentive compatible?

- Kroll, Davey, Felten. WEIS, 2013. Economics of Bitcoin mining.

Yes, assuming all parties are rational, and strategy space is limited

- Eyal and Sirer, 2013. Bitcoin is Vulnerable, Majority is Not Enough. arXiv

No, assuming all parties are rational, slightly larger strategy space, at least ⅓ is controlled by a single entity

Incentive compatibility

(Mixed) Strategy - a (randomized) program to run

Preference - defined over possible outcomes (or prospects)

Equilibrium - given knowledge of other players strategies, is the current strategy preferable?

Incentive compatible - the honest strategy is an equilibrium

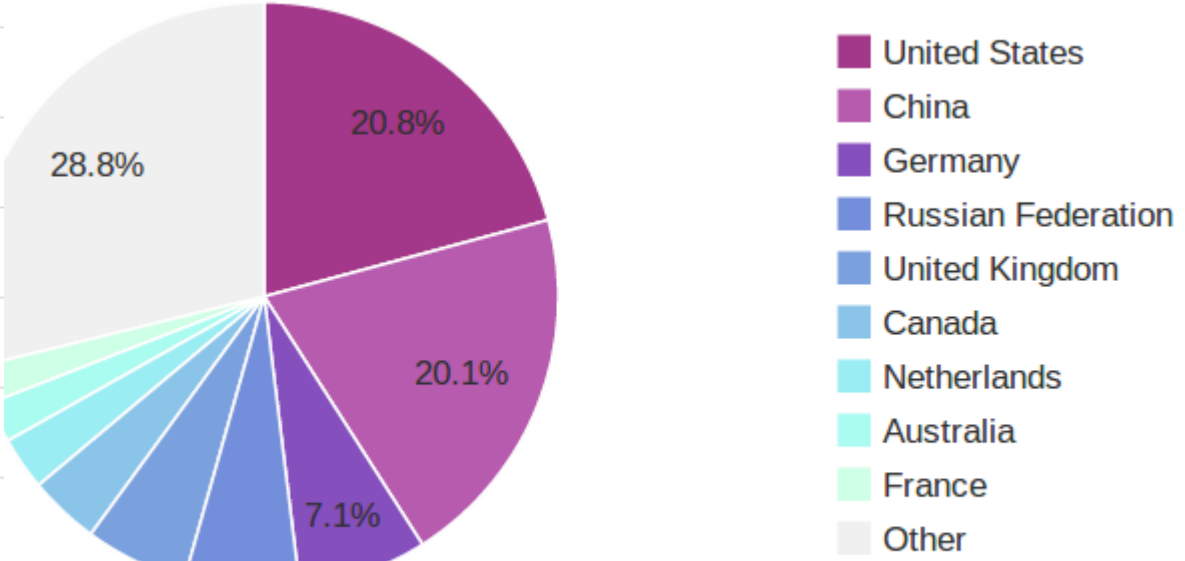
Interlude: Current Events

as of December 2013

Price 2013-2014

getaddr.bitnodes.io - 2013 Addy Yeow

1	United States	50484
2	China	48863
3	Germany	17232
4	Russian Federation	15334
5	United Kingdom	13721
6	Canada	9416
7	Netherlands	7077
8	Australia	5469
9	France	5024
10	Poland	4706



Geographic distribution of nodes (as of Dec 2013)

Black Markets

Silk Road $14M revenue (estimated) in 2012

Shut down in Sep. 2013, founder arrested

Silk Road 2.0 appears 2 weeks later

Sheep Market announces $6M theft, closes

Black Market Reloaded closes gracefully

Cristin. WWW 2013. Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace.

ADS Case Study: Bitcoin

Recently popular peer-to-peer virtual currency.

Features a hash-based ADS representing a log of transactions and a ledger of account balances. (Integrity, *not* privacy)

We can use λ• to model the existing algorithm...

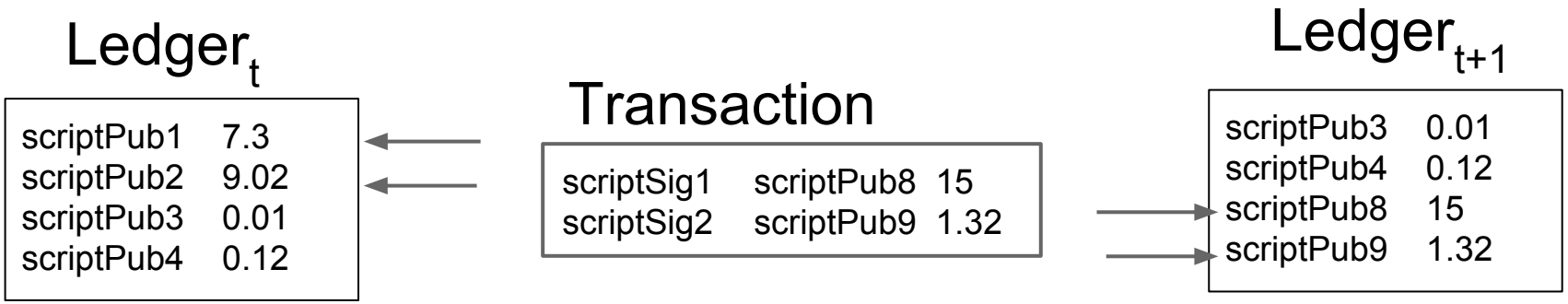
and propose an optimization

How Transactions Work

The ledger actually maps quantities of BTC to Access Control Policy scripts.

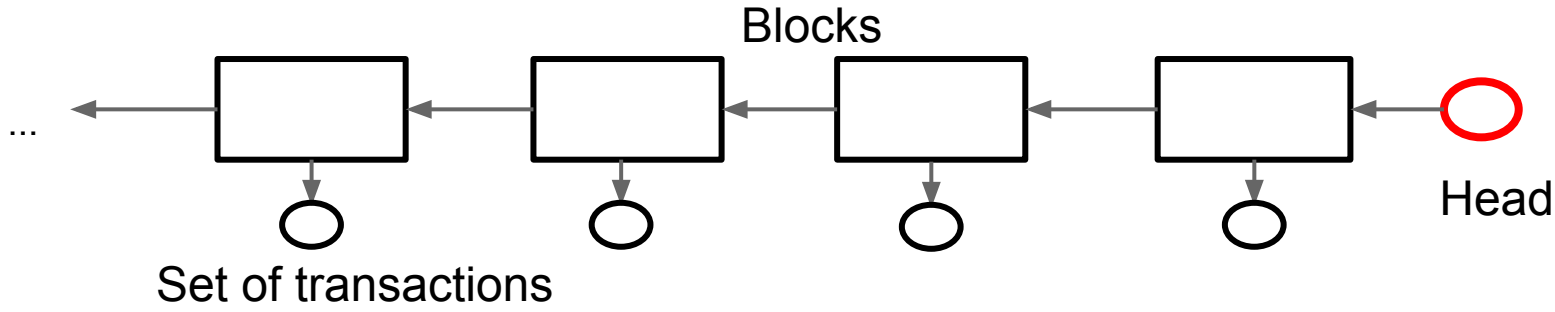
ScriptSig (Witness)	ScriptPubKey (Statement)
{signature}	{pubkey} OP_CHECKSIG
{signature} {pubkey}	OP_DUP OP_HASH {h(pubkey)} OP_EQUAL OP_CHECKSIG
{signature_1} … {signature_m}	m {pubkey_1}...{pubkey_n} n OP_CHECKMULTISIG

How Transactions Work



“Best Practices” implemented by standard client: Create a new keypair for every transaction.
Neither **scriptPub8** nor **scriptPub9** resemble **scriptPub1** or **scriptPub2**
However, we would infer that 1.32 is a “change” transaction, because 15 is a round number.
Thus **scriptPub1**, **scriptPub2**, and **scriptPub9**, all likely belong to same user.

The Bitcoin Block Chain




```
type coin = int
type transaction =
      coin list (* coins to remove *) ×
      coin list (* coins to insert *)
type ledger = IntSet.t (* Built-in set *)
type block = Genesis | Block of • block × • transaction
```

```
let apply tx ldgr =
  let after_remove = List.fold_right
    (IntSet.remove) (fst tx) ldgr in
  let after_insert = List.fold_right
    (IntSet.add) (snd tx) after_remove in
  after_insert
```

Every client has to store entire ledger!
Storage cost: O(m), where m is size of ledger

Optimized Bitcoin Validation

```
type coin = int
type transaction =
      coin list (* coins to remove *) ×
      coin list (* coins to insert *)
type ledger2 = • Redblack.tree
let apply2 tx ldgr : ledger2 =
   let after_remove =
      List.fold_right (Redblack.delete) (fst tx) ldgr
   in let after_insert =
      List.fold_right (Redblack.add) (snd tx) after_remove
   in after_insert
```

Use authenticated ledger
(e.g., RedBlack+ tree) instead.
Storage cost: O(1)

Conclusions

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

Implementation available: amiller.github.io/lambda-auth

Future work: incorporate stronger cryptographic primitives

Zero Knowledge - privacy, not just integrity

SNARKs - compression for computation, not just data