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?

Untrusted Server
Plenty of storage

Trusted Client
Not much storage

Queries
Responses

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

Step 1: fetch(2)

Step 2: compute proof, or Verification Object (VO)

Step 3: verify by recomputing hashes

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 $\lambda\bullet$ ("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

- 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 $\tau"), coercions
  - $auth: \tau \rightarrow \bullet\tau$
  - $unauth: \bullet\tau \rightarrow \tau$
- Evaluation mode for prover, verifier.
  - Prover produces VO, verifier consumes/checks it
  - Result should relate to "ideal" mode

```ocaml
type tree =
|   Tip
|   Bin of tree $\times$ int $\times$ tree

let rec member (t:tree) (x:int) : bool =
  match 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
```
Ideal mode: Data and operations

• Authenticated types are the identity

  type \( \bullet \tau = \tau \)

  \text{auth} \ x = x

  \text{unauth} \ x = x

• As such, easy to reason about what ADS is doing
Prover and Verifier: Data

At Prover, a value of type $\bullet \tau$ is $\langle d, v \rangle$ where

- value $v$ has type $\tau$
- $d$ 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 $\bullet \tau$ is a hash $d$
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... \( \lambda \)

\( \lambda \): a simple (turing equiv.) computing model

Terms := \( \lambda x. e \mid e \ e' \mid x \)

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

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.
Types: syntactic classes of programs

\[
\text{Types } \tau ::= 1 | \tau_1 \rightarrow \tau_2 | \tau_1 + \tau_2 | \tau_1 \times \tau_2 | \mu \alpha.\tau | \alpha | \bullet \tau
\]

Rules for program composition:

\[
\frac{\Gamma \vdash v : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash v' : \tau_1}{\Gamma \vdash vv' : \tau_2}
\]

\[
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}
\]

\[
\frac{\Gamma \vdash v : \tau_1}{\Gamma \vdash \text{inj}_1 v : \tau_1 + \tau_2}
\]

\[
\frac{\Gamma \vdash v : \tau_2}{\Gamma \vdash \text{inj}_2 v : \tau_1 + \tau_2}
\]

Type soundness:
Reduction preserves types

and so on....
PL for Crypto people... editorial

Why another computing model?

1. Better fits the computational model
   - Typed $\lambda$ 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 $\rightarrow$ 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

Types $\tau$ ::= $1$ | $\tau_1 \rightarrow \tau_2$ | $\tau_1 + \tau_2$ | $\tau_1 \times \tau_2$ | $\mu \alpha. \tau$ | $\alpha$

Values $v$ ::= $(\cdot)$ | $x$ | $\lambda x. e$ | $\text{rec } x. \lambda y. e$

| $\text{inj}_1 v$ | $\text{inj}_2 v$ | $(v_1, v_2)$ | $\text{roll } v$

Exprs $e$ ::= $v$ | $\text{let } x = e_1 \text{ in } e_2$ | $v_1 \ v_2$ | $\text{case } v \ v_0 \ v_1$

| $\text{prj}_1 v$ | $\text{prj}_2 v$ | $\text{unroll } v$ | $\text{auth } v$ | $\text{unauth } v$

\[
\frac{\Gamma \vdash v : \tau_1}{\Gamma \vdash \text{inj}_1 v : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash \text{inj}_2 v : \tau_1 + \tau_2}{\Gamma \vdash \text{case } \Gamma \vdash v_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash v_2 : \tau_2 \rightarrow \tau}$

\[
\frac{\Gamma \vdash v : \tau_1 + \tau_2}{\Gamma \vdash \text{unauth } v : \tau} \quad \frac{\Gamma \vdash \text{auth } v : \bullet \tau}{\Gamma \vdash \text{unauth } v : \tau} \quad \frac{\Gamma \vdash \text{auth } v : \bullet \tau}{\Gamma \vdash \text{unauth } v : \tau}
\]
Operational Semantics in 3 Modes

\[ \Gamma \vdash e : \tau \]

Input program

“Compilation”

\[ e \rightarrow_{m} e_{P} \rightarrow_{V} \]

- \( m \) is either I, P, or V
- carry around the VO \( \pi \)
- most transitions leave it unchanged

\[ \langle \pi, \text{case } (\text{inj}_1 v)(\lambda x.e_1)(\lambda x.e_2) \rangle \rightarrow_{m} \langle \pi, e_1[v/x] \rangle \]

\[ \langle \pi, \text{case } (\text{inj}_2 v)(\lambda x.e_1)(\lambda x.e_2) \rangle \rightarrow_{m} \langle \pi, e_2[v/x] \rangle \]

\[ \langle \pi, \text{prj}_1 (v_1, v_2) \rangle \rightarrow_{m} \langle \pi, v_1 \rangle \]

\[ \langle \pi, \text{prj}_2 (v_1, v_2) \rangle \rightarrow_{m} \langle \pi, v_2 \rangle \]

... 

\[ \langle \pi, \text{auth } v \rangle \rightarrow_{I} \langle \pi, v \rangle \]

\[ \langle \pi, \text{unauth } v \rangle \rightarrow_{I} \langle \pi, v \rangle \]

auth/unauth are no-ops in ideal mode
Operational Semantics in 3 Modes

Shallow projection function, written \(|e|\)

\[
\begin{align*}
\langle () \rangle & = () \\
\langle (h, v) \rangle & = h \\
\langle \text{auth } v \rangle & = \text{auth } (v) \\
\langle (v_1, v_2) \rangle & = ((v_1), (v_2)) \\
\langle \text{roll } v \rangle & = \text{roll } (v) \\
\langle \text{rec } x.\lambda y.e \rangle & = \text{rec } x.\lambda y.e
\end{align*}
\]

\[
\begin{align*}
\ll \pi, \text{auth } v \gg 
& \rightarrow_P \ll \pi, \langle \text{hash } (v), v \rangle \gg \\
\ll \pi, \text{auth } v \gg 
& \rightarrow_V \ll \pi, \text{hash } v \gg
\end{align*}
\]

\[
\begin{align*}
\ll \pi, \text{unauth } (h, v) \gg 
& \rightarrow_P \ll \pi \circ [ (v) ], v \gg \\
\text{hash } s_0 & = h \\
\ll [s_0] \circ \pi, \text{unauth } h \gg & \rightarrow_V \ll \pi, s_0 \gg
\end{align*}
\]

auth in Prover/Verifier builds new digest

unauth in Prover adds to the stream

unauth in Verifier consumes from the stream
3-Way Agreement Relation

most terms preserve self-agreement

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 $\ll [\ ] , e \rr \rightarrow^i_I \ll [\ ] , e' \rr$, then we can run Prover, and give its output to Verifier, get correct answer.

Security: If for a possibly malicious prover, $\ll [\pi_A] , e_V \rr \rightarrow^i_V \ll [\pi'] , e'_V \rr$ then either:

( verifier is correct)

\[
\ll [\ ] , e \rr \rightarrow^i_I \ll [\ ] , e' \rr
\ll [\ ] , e_P \rr \rightarrow^i_P \ll [\pi] @ \pi' , e'_P \rr
\pi_A = \pi @ \pi'
\vdash e' \ e'_P \ e'_V : \tau
\]

or:

(we can find a hash collision)

\[
j \leq i,
\ll [\ ] , e_P \rr \rightarrow^j_P \ll [\pi_0] @ [s] , e'_P \rr
\pi_A = \pi_0 @ [s^\dagger] @ \pi'
\]

$s \neq s^\dagger$ but hash $s = \text{hash } s^\dagger$. 
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

```plaintext

type \( \bullet \alpha = \) Digest of string (* the digest *)
  | Prover of string \( \times \) \( \alpha \)

let auth_prover (shallow: \( \alpha \rightarrow \alpha \)) (v:\( \alpha \)) : \( \bullet \alpha = \)
  Prover(hash (shallow v), v)

let unauth_prover (shallow: \( \alpha \rightarrow \alpha \)) (v:\( \bullet \alpha \)) : \( \alpha = \)
  let Prover(_,x) = v in
to_channel !prf_output (shallow x);

let auth_verifier (v:\( \alpha \)) : \( \bullet \alpha = \) Digest(hash v)

let unauth_verifier (v:\( \bullet \alpha \)) : \( \alpha = \)
  let Digest(h) = v in
  let y = from_channel !prf_input in
  assert h = hash y;
y

let shallow_(\bullet) (Prover(h,_) : \( \bullet \alpha \)) : \( \bullet \alpha = \) Digest(h)

(* User-provided code *)

type bst = Tip
  | Bin of \( \bullet \)bst \( \times \) int \( \times \) \( \bullet \)bst
  | AuthBin of \( \bullet \) (bst \( \times \) int \( \times \) bst)

let is_empty (t:\( \bullet \)bst) : bool = (unauth t = Tip)
let mk_leaf (x:int) : \( \bullet \)bst = AuthBin(auth(Tip, x, Tip))

(* Generated Prover code *)

let rec shallow_bst : bst \( \rightarrow \) bst = function
  | Tip \( \rightarrow \) Tip
  | Bin (x, y, z) \( \rightarrow \) Bin(shallow_\( \bullet \) x, y, shallow_\( \bullet \) z)
  | AuthBin (x) \( \rightarrow \) AuthBin (shallow_bst1 x)

and shallow_bst1 : bst \( \times \) int \( \times \) bst \( \rightarrow \) bst \( \times \) int \( \times \) bst
  = function (x, y, z) \( \rightarrow \) (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

```ocaml
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 \textit{auth} and \textit{unauth}
- 30% in Marshal
- Tags?

Prover:
- 28% in SHA1
- used in \textit{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
3. Suppose we have a magic token that chooses parties at random. Whoever has the token gets to broadcast *once*

If $t$ parties are malicious, $\Pr[\text{honest selected}] = \frac{n-t}{t}$

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

\[
\text{Scratch}_d(puz, m): \ r \leftarrow \{0, 1\}^k; \ \text{if } H(puz || m || r) < 2^{k-d} \text{ then return } r
\]
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
     $|\text{chain}| > |\text{preferred}|$ then
    preferred := chain

mainloop(): (as fast as possible)
  puzz := $H(\text{preferred})$
  r ← Scratch(puzz, 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:
Is it actually incentive compatible?

  
  **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 $\frac{1}{3}$ 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
### Geographic Distribution of Nodes (as of Dec 2013)

<table>
<thead>
<tr>
<th>Country</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>United States</td>
<td>50484</td>
</tr>
<tr>
<td>China</td>
<td>48863</td>
</tr>
<tr>
<td>Germany</td>
<td>17232</td>
</tr>
<tr>
<td>Russian Federation</td>
<td>15334</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>13721</td>
</tr>
<tr>
<td>Canada</td>
<td>9416</td>
</tr>
<tr>
<td>Netherlands</td>
<td>7077</td>
</tr>
<tr>
<td>Australia</td>
<td>5469</td>
</tr>
<tr>
<td>France</td>
<td>5024</td>
</tr>
<tr>
<td>Poland</td>
<td>4706</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Percentage Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>United States</td>
</tr>
<tr>
<td>China</td>
</tr>
<tr>
<td>Germany</td>
</tr>
<tr>
<td>Russian Federation</td>
</tr>
<tr>
<td>United Kingdom</td>
</tr>
<tr>
<td>Canada</td>
</tr>
<tr>
<td>Netherlands</td>
</tr>
<tr>
<td>Australia</td>
</tr>
<tr>
<td>France</td>
</tr>
<tr>
<td>Poland</td>
</tr>
</tbody>
</table>

*getaddr.bitnodes.io - 2013 Addy Yeow*
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.
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 $\lambda\bullet$ to model the existing algorithm... and propose an optimization
How Transactions Work

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

<table>
<thead>
<tr>
<th>ScriptSig (Witness)</th>
<th>ScriptPubKey (Statement)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{signature} {pubkey}</td>
<td>{pubkey} OP_CHECKSIG</td>
</tr>
<tr>
<td>{signature} {pubkey}</td>
<td>OP_DUP OP_HASH {h(pubkey)} OP_EQUAL OP_CHECKSIG</td>
</tr>
<tr>
<td>{signature_1} ... {signature_m}</td>
<td>m {pubkey_1}...{pubkey_n} n OP_CHECKMULTISIG</td>
</tr>
</tbody>
</table>
“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

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

```
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
```
Optimized Bitcoin Validation

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