LBTrust: Declarative Reconfigurable Trust Management

Bill Marczak*, Dave Zook†, Wenchao Zhou*, Molham Aref†, Boon Thau Loo*

*University of Pennsylvania
†LogicBlox
What is “Trust Management”?

- Trust management is broadly defined as:
  - Assigning credentials (rights) to principals (users) to perform actions
  - Delegating among principals
  - Enforce access control policies in a multi-user environment

- Logic representation/reasoning:
  - Logical analysis of new security protocols
  - Declarative interface for implementing security policies
  - Several runtime systems based on distributed Datalog/Prolog

- Binder, a simple representative language:
  At alice:
  
  r1: access(P,O,read) ← good(P).
  r2: access(P,O,read) ← bob says access(P,O,read).

  “In alice's context, any principal P may access object O in read mode if P is good (R1) or, bob says P may do so (R2 - delegation)”
(Non-Exhaustive) Survey of Trust Management Languages

<table>
<thead>
<tr>
<th></th>
<th>Authentication</th>
<th>Delegation</th>
<th>Conditional Re-Delegation</th>
<th>Threshold Structures</th>
<th>Type System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aura</td>
<td>Y</td>
<td>Y*</td>
<td>Y?</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Binder</td>
<td>Y</td>
<td>Y*</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Cassandra</td>
<td>Y</td>
<td>Y*</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>D1LP</td>
<td>Y</td>
<td>Y</td>
<td>Y (depth/width)</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>KeyNote</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>SD3</td>
<td>Y</td>
<td>Y*</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SeNDLoG</td>
<td>Y</td>
<td>Y*</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>SPKI/SDSI</td>
<td>Y</td>
<td>Y*</td>
<td>Y (boolean)</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

- Problem: too many languages, features, separate runtime systems, hard to compare and reuse
- Our goal: A unified declarative framework to enable all of these languages
Key Ideas of LBTrust

- Constraints: type safety, program correctness, security
- Meta-programmability
  - Meta-model: rules as data [VLDB 08]
  - Meta-rules (code generation)
  - Meta-constraints (constraint + reflection)
- Customizable partitioning, distribution, and communication
- Extensible predicates for cryptographic primitives
Constraints and Types

fail() ← access(P,O,M), !principal(P).

“let fail() whenever access(P,O,M) and not principal(P)”

access(P,O,M) → principal(P).

“whenever access(P,O,M), require principal(P)”

access(P,O,M) → principal(P), object(O), mode(M).

type constraint
Meta-Model Schema

rule(R) \rightarrow .
active(R) \rightarrow rule(R).
head(R,A) \rightarrow rule(R), atom(A).
body(R,A) \rightarrow rule(R), atom(A).

atom(A) \rightarrow .
functor(A,P) \rightarrow atom(A), predicate(P).
arg(A,I,T) \rightarrow atom(A), int(I), term(T).
negated(A) \rightarrow atom(A).

term(T) \rightarrow .
variable(X) \rightarrow term(X).
vname(X,N) \rightarrow variable(X), string(N).
constant(C) \rightarrow term(C).
value(C,V) \rightarrow constant(C), string(V).

predicate(P) \rightarrow .
pname(P,N) \rightarrow predicate(P), string(N).

ensures rules are well-structured
Rules as Data

foo(x) ← bar(x).

“let foo(x) whenever bar(x)”
Meta Rules for Security

- Meta
  - Code generation (insert new rules that must be evaluated)
  - Reflection (query for program structure)
- Meta-Syntax
  - Embedded rule/bounded constants

\[
\text{active}([\mid \text{active}(R) \leftarrow \text{says}(\neg P_2, \neg P_1, R). \mid]) \leftarrow \text{delegates}(P_1, P_2).
\]

“activate a rule \( \text{active}(R) \leftarrow \text{says}(P_2, P_1, R) \). for every delegates(P1,P2).”
Meta-Constraints

- Meta
  - Code generation (insert new rules that must be evaluated)
  - Reflection (query for program structure)

```
owner(U, [ | A <- P(T*), A*. | ]) → access(U,P,read).
```

“Whenever user U owns a rule, require that U has read access to every predicate P in the rule body”

```
fail() ← owner(U, [ | A <- ~P(T*), A*. | ]), !access(U,P,read).
```

- Meta variables
- Kleene star
A Concrete Example: The “Says” Authentication Construct

\[
\text{says}(P_1, P_2, R) \rightarrow \text{prin}(P_1), \text{prin}(P_2), \text{rule}(R).
\]
\[
\text{rulesig}(R, S) \rightarrow \text{rule}(R), \text{string}(S).
\]
\[
\text{rsapubkey}(P, K) \rightarrow \text{prin}(P), \text{string}(K).
\]
\[
\text{rsaprivkey}(P, K) \rightarrow \text{prin}(P), \text{string}(K).
\]

\[
\text{rsasign}(R, S, K).
\]

\[
\text{rsaverify}(R, S, K).
\]
Delegation (Basic)

alice “speaks-for” bob == “if alice says something, bob says it too.”

*speaks-for* is a special form of delegation:
- delegates(P1,P2) \(\rightarrow\) prin(P1), prin(p2).

\[ \text{delegates}(\text{bob}, \text{alice}). \]

“I will believe (i.e. say) any rule that alice says”

\[ \text{says} (\text{alice}, \text{bob}, R). \]

\[ \text{r1: active([ | active(R) \leftarrow \text{says}(P2, P1, R). | ])} \leftarrow \text{delegates}(P1, P2). \]

\[ \text{r2: active(R) \leftarrow says(alice, bob, R).} \]
Other cool features (see paper for details)

- Conditional Delegations:
  - Constraint by width, depth, or predicates
  - Detecting delegation violations (use of provenance)
- Customizable distribution/partitioning policies
  - Partition data and rules by principals
  - Distribute principals across machines
  - Same security policy rules can run in local/distributed environment
- Customizable authentication and encryption (RSA vs HMAC)
- Use meta-rules to rewrite top-down access control to execute in a bottom-up evaluation engine
- Example languages:
  - Binder
  - Delegation logic, D1LP
  - Secure Network Datalog [ICDE 09]
    - Authenticated routing protocols
LogicBlox - a commercial Datalog Engine

• Startup company based in Atlanta (50 employees + 65 academic collaborators)
• Decision Automation Applications:
  ◦ Retail supply-chain management (Predictix) - e.g: Best Buy, Sainsbury,
  ◦ Insurance risk management (Verabridge) - e.g. RenRe
  ◦ Context Sensitive Program Analysis (Semmle) - TBD
• LBTrust is developed using LogicBlox:
  ◦ Classic datalog with well behaved constructors or E variables in head
  ◦ Constraints
  ◦ Meta-programmability: model, rules, constraints
  ◦ Higher-Order: gets us aggs, state + ECA, default values, etc.