AURA:
A language with authorization and audit

Steve Zdancewic

University of Pennsylvania

WG 2.8  2008
Security-oriented Languages

Limin Jia, Karl Mazurak, Jeff Vaughan, Jianzhou Zhao, Joey Schorr, and Luke Zarko

- Manifest Security Project (NSF-0714649)
  - Penn: Benjamin Pierce, Stephanie Weirich
  - CMU: Karl Crary, Bob Harper, Frank Pfenning

- CAREER: Language-based Distributed System Security (NSF-0311204)
Goal of the AURA project:

• Develop a security-oriented programming language that supports:
  – Proof-carrying Authorization
    [Appel & Felton] [Bauer et al.]
  – Strong information-flow properties
    (as in Jif [Myers et al.], FlowCaml [Pottier & Simonet])

• Why?
  – Declarative policies (for access control & information flow)
  – Auditing & logging: proofs of authorization are informative
  – Good theoretical foundations

• In this talk: tour of AURA's
  – Focus on the authorization and audit components
Outline

• AURA's programming model

• Authorization logic
  – Examples

• Programming in AURA
  – (Restricted) Dependent types

• Status, future directions, conclusions
AURA is a call-by-value type-safe functional programming language.

As in Java, C#, etc., AURA provides an interface to the OS resources:
- disk, network, memory, …

AURA is intended to be used for writing security-critical components.
AURA: Authorization Policies

- AURA security policies are expressed in an authorization logic
- Applications can define their own policies
- Language provides features for creating/manipulating proofs
AURA: Authorization Policies

- Proofs are first class and they can depend on data
- Proof objects are capabilities needed to access resources protected by the runtime: AURA's type system ensures compliance
- The runtime logs the proofs for later audit
AURA: Principals and Keys

- For distributed systems, AURA also manages private keys
- Keys can create policy assertions sharable over the network
- Connected to the policy by AURA's notion of principal
Evidence-based Audit

• Connecting the contents of log entries to policy helps determine *what* to log.
Evidence-based Audit

• Connecting the contents of log entries to policy helps determine *what* to log.

• Proofs contain structure that can help administrators find flaws or misconfigurations in the policy.
Evidence-based Audit

- Connecting the contents of log entries to policy helps determine *what* to log.

- Proofs contain structure that can help administrators find flaws or misconfigurations in the policy.

- Reduced TCB: Typed interface forces code to provide auditable evidence.
Outline

• AURA's programming model

• Authorization logic
  – Examples

• Programming in AURA
  – (Restricted) Dependent types

• Status, future directions, conclusions
AURA's Authorization Logic

• Policy propositions
  \( \varphi ::= \text{true} \)

\[ \begin{align*}
  c \\
  A \text{ says } \varphi \\
  \alpha
\end{align*} \]

• Principals
  A, B, C … P, Q, R etc.

• Constructive logic:
  – proofs are programs
  – easy integration with software

• Access control in a Core Calculus of Dependency
  [Abadi: ICFP 2006]

  Encoded using \( \Pi \) types and inductive datatypes.
Example: File system authorization

- **P1**: FS says (Owns A f1)
- **P2**: FS says (Owns B f2)
- ...

- **OwnerControlsRead**: 
  FS says \( \forall o, r, f \cdot (\text{Owns } o \ f) \rightarrow \) 
  \( (o \ \text{says} \ (\text{MayRead } r \ f)) \rightarrow \) 
  \( (\text{MayRead } r \ f) \)

- Might need to prove: FS says (MayRead A f1)
- What are "Owns" and "f1"?
Decentralized Authorization

• Authorization policies require application-specific constants:
  – e.g. "MayRead B f" or "Owns A f"
  – There is no "proof evidence" associated with these constants
  – Otherwise, it would be easy to forge authorization proofs

• But, principal A should be able to create a proof of A says (MayRead B f)
  – No justification required -- this is a matter of policy, not fact!

• Decentralized implementation:
  – One proof that "A says T" is A's digital signature on a string "T"
  – written \( \text{sign}(A, "T") \)
Example Proof (1)

- **P1**: FS says (Owns A f1)
- **OwnerControlsRead**: 
  FS says  \( \forall o,r,f. (\text{Owns } o \ f) \rightarrow (o \text{ says (MayRead } r \ f)) \rightarrow (\text{MayRead } r \ f) \)

---

- Direct authorization via FS's signature:

  \[ \text{sign(FS, "MayRead A f1")} \]
  \[ : \text{FS says (MayRead A f1)} \]
Example Proof (2)

- **P1**: FS says (Owns A f1)
- **OwnerControlsRead**: 
  
  FS says  \( \forall o, r, f. \ (\text{Owns } o \ f) \rightarrow (o \text{ says (MayRead } r \ f)) \rightarrow (\text{MayRead } r \ f) \)

- Complex proof constructed using "bind" and "return"

  ```
  bind p = OwnerControlsRead in 
  bind q = P1 in 
  return FS (p A A f1 q sign(A,"MayRead A f1"))) 
  : FS says (MayRead A f1)
  ```
Authority in AURA

• How to create the value $\text{sign}(A, "\varphi")$?

• Components of the software have authority
  – Authority modeled as possession of a private key
  – With A's authority:
    $$\text{say}("\varphi") \text{ evaluates to } \text{sign}(A, "\varphi")$$

• What $\varphi$'s should a program be able to say?
  – From a statically predetermined set (static auditing)
  – From a set determined at load time

• In any case: log which assertions are made
Outline

• AURA's programming model

• Authorization logic
  – Examples

• Programming in AURA
  – (Restricted) Dependent types

• Status, future directions, conclusions
**AURA Programming Language**

**Types:** describe programs
- `int` FileHandle
- `string` prin
- `int -> int` pf $\varphi$

**Propositions:** specify policy
- $\varphi$ A says $\varphi$
- $(\varphi \land \phi)$ $\forall \alpha. T$
- $(\text{Owns } A \ fh1)$ $(\varphi \rightarrow \phi)$

**Programs:** computations, I/O
- 3 fh1
- "hello" A
- `say($\varphi$)` $\backslash x:t.e$

**Evidence:** proofs/credentials
- `sign(A, "\varphi")`
- `bind/return $\backslash x:t.e$`

**Static**

**Dynamic**
(Restricted) Dependent Types

• Policy propositions can mention program data
  – E.g. "f1" is a file handle that can appear in a policy
  – AURA restricts dependency to first order data types
  – Disallows computation at the type level – only values!

• Programming with dependent types:

  \( \{ x : T ; \ U(x) \} \)  
  \( (x : T) \rightarrow U(x) \)  

  dependent pair*  
  dependent functions

• Invariant: sign only types
  – Computation can't depend on signatures
  – But, can use predicates:  \( \{ x : \text{int}; \ pf A \ says \ Good(x) \} \)
Auditing Interfaces

• Type of the "native" read operation:
  \[
  \text{raw\_read} : \text{FileHandle} \rightarrow \text{String}
  \]

• AURA's runtime exposes it this way:
  \[
  \text{read} : (f:\text{FileHandle}) \rightarrow \\
  \text{pf RT says (OkToRead self f)} \rightarrow \\
  \{\text{ans:} \text{String}; \text{pf RT says (DidRead f ans)}\}
  \]

• RT is a principal that represents the AURA runtime
• OKtoRead and DidRead are "generic" policies
  – The application implements its own policies about when it is OKtoRead by providing assertions, etc.
  – Parts of the runtime must delegate to the application
Signatures

• Assertions: uninhabited constants that construct Prop’s

```
assert MayRead : Prin -> FileHandle -> Prop;
assert Owns : Prin -> FileHandle -> Prop;
```

• AURA supports mutually recursive datatypes and mutually inductively defined propositions:

```
data List: Type -> Type {
    | nil : (t:Type) -> List t
    | cons: (t:Type) -> t -> List t -> List t
}
data OwnerInfo : FileHandle -> Type {
    | oinfo : (f:FileHandle) -> (p:Prin)
        -> pf (self says (Owns p f)) -> OwnerInfo f
}
data And : Prop -> Prop -> Prop {
    | both : (p:Prop) -> (q:Prop) -> p -> q -> And p q
}
```
More about Prop vs. Type

• We want the Prop fragment to be a logic:
  – Pure, strongly normalizing
  – Signature typing rules add a strong positivity constraint for Prop to rule out divergence

• We need to separate the Prop and Type fragments
  – Type fragment includes divergent terms (possibly other effects)
  – This is the purpose of the “pf” monad. A value of type “pf P” is of the form “returnₚ t” where “t” is a pure proof term that proves P.
  – It is possible to write a loop of type “pf P” by not one of type “P”.
Example Program

• (see demo.core)
Formalizing Core AURA

• Lambda-cube-like representation with a very simple core:

\[
\begin{align*}
t ::= & \ x \mid \text{ctr} \mid \lambda x: t_1. t_2 \mid t_1 \ t_2 \mid (x: t_1) \rightarrow t_2 \mid \\
& \text{match } t_1 \ t_2 \text{ with } \{b\} \mid (t_1 : t_2) \mid c
\end{align*}
\]

• Plus these constants (special typechecking rules):

\[
c ::= \text{Type} \mid \text{Prop} \mid \text{Kind} \\
prin \mid \text{says} \mid \text{return}_s \mid \text{bind}_s \\
self \mid \text{sign} \\
pf \mid \text{return}_p \mid \text{bind}_p \\
if
\]
Coq Formalization

- Type system and operational semantics:
  - 30 rules in 4 mutually inductive predicates: wf_env, wf_tm, wf_branches, wf_brn
  - Signature checking: wf_sig, wf_bundle_tcrs, wf_bundle_ctrs, wf_ctr_decls
  - Conversion relation (for casts) that reflects dynamic equality checks into the static type system
  - Evaluation rules

- Correctness properties proved in Coq:
  - Type soundness and decidability of typechecking (~7000 loc)
  - Decidability of typechecking is simplified by:
    - Restricted dependency (only values)
    - Limited equality proofs available statically

- Paper proof of strong normalization of (a slightly simplified version of) the Prop fragment.
Observations about the Formalization

• Dealing with mutually recursive datatypes and pattern matching was a *lot* of work
  – Significant source of complexity for soundness and decidability
  – … hopefully reusable in other contexts (our lambda cube plus constants can probably be instantiated to other languages)

• Initial investment in formalization was heavy – many hours to implement the typing rules, etc.
  – But: having machine checked proofs is a big win, especially for large groups of collaborators.
  – It gets easier over time…
Open Questions

• **AURA** needed improvements:
  - Anonymous existential types / dependent type & inference
  - Richer dependent types?
  - Explicit / richer equality proofs?
  - Revocation/expiration of signed objects? [Garg and Pfenning]
  - Connection to program verification?
  - Correlate distributed logs?

• This story seems just fine for *integrity*, but what about *confidentiality*?
  - We have many ideas about connecting to information-flow analysis
  - Is there an "encryption" analog to "signatures" interpretation?
  - Encode confidentiality using “security monads” [work at Chalmers]
Conjecture: Non-security use?

- Carve up a program into principals
  - Perhaps by module?
- Allow principals to make arbitrary (dependent) logical assertions
  - Interfaces can specify constraints in this logic
  - (e.g. propositions regulate type equality)
- The “says” modality offers an escape hatch: no need to construct an actual proof
  - Cast uses “asserted equality” (not “verifiable equality”)
  - “says” isolates components, allows assignment of *blame* and makes trust relationships explicit.
- Question: is this interesting? Useful? Does anyone know of any work similar to this?
Outline

• AURA's programming model

• Authorization logic
  – Examples

• Programming in AURA
  – Dependent types

• Status, future directions, conclusions
AURA's Status

• Have implemented an interpreter in F#
  – Many small examples programs
  – Working on larger examples
  – Goal: experience with proof sizes, logging infrastructure

• Planning to compile AURA to Microsoft .NET platform
  – Proof representation / compatibility with C# and other .NET languages
  – Luke Zarko is awesome
    • Penn undergrad applying this fall to Ph.D. programs for next year
AURA

- A language with support for authorization and audit
- Authorization logic
- Limited form of dependent types
- Language features that support secure systems

www.cis.upenn.edu/~stevez/sol
Thanks!