Evidence-based Audit

Jeff Vaughan
Limin Jia, Karl Mazurak, and Steve Zdancewic

Department of Computer and Information Science
University of Pennsylvania

IBM PL Day/NJPLS
August 28, 2008
Our Setting: Distributed Access Control

Diagram:
- Application
- Resource
- Principal

Legend:
- Application
- Resource
- Principal
Our Setting: Distributed Access Control
Our Setting: Distributed Access Control
Our Setting: Distributed Access Control

Application
Resource
Principal
Data

Diagram:

- Application
- Data
- Resource
- Principal
Our approach: Proofs attest to message validity.
Augmenting requests with logged evidence (proofs) enables principled access control and meaningful audit in distributed systems.
The Aura Project

Key Idea

Augmenting requests with logged evidence (proofs) enables principled access control and meaningful audit in distributed systems.

- A programming language called Aura
  - A propositional fragment: the evidence
  - An ML-like computation language
- A security aware programming model
  - Active, potentially malicious principals
  - Mutual distrust between applications and principals
  - Emphasis on access control and audit
- An implementation
  - Mechanized Coq proofs
  - A prototype interpreter and .Net-based runtime
The Aura Project

Key Idea

Augmenting requests with logged evidence (proofs) enables principled access control and meaningful audit in distributed systems.

- A programming language called Aura
  - A propositional fragment: the evidence
  - An ML-like computation language
- A security aware programming model
  - Active, potentially malicious principals
  - Mutual distrust between applications and principals
  - Emphasis on access control and audit
- An implementation
  - Mechanized Coq proofs
  - A prototype interpreter and .Net-based runtime
Security Problem

An application may contain bugs or be configured with incorrect formal rules.

Aura Solution

Trust only a small *kernel* that isolates applications and resources. Log proofs corresponding to all access control decisions.

- [Saltzer+ 75], [Bauer+ 99], [Jia+ 08]
- [Wee 95], [Cederquist+ 05]
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.

If \((\text{op2} \blacktriangleleft \blacklozenge)\) is well-typed then (forward \(\blacktriangleleft\) to resource; log \(\{\blacktriangleleft, \blacklozenge\}; \ldots\)) else skip

\[
\begin{align*}
\text{Kernel, } K \\
\text{Log} \\
\text{raw-op1} \\
\text{raw-op2} \\
\text{Resource}
\end{align*}
\]
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
In Aura, a lightweight kernel protects resources.
Example: Types for playing an audio file.

The current runtime kernel: Kernel : prin.

Raw operation generating a world effect:

\[
\text{playWav} : (s: Song) \rightarrow Unit
\]

Exposed interface operation wrapping \text{playWav}:

\[
\text{playFor} : (s: Song) \rightarrow (p: prin) \rightarrow \text{pf (Kernel says (MayPlay p s))} \rightarrow Unit
\]

The function’s third argument is an access control proof.
Security Problem

The formal rules language may be too impoverished to express institutional policy.

Aura Solution

Use dependent DCC with signature objects to specify rules.

[Abadi+ 06], [Fournet+ 07], [Bengtson+ 08]
Aura syntax represents both proofs and computations.

**Partial syntax**

- $t ::= \text{Prop} | \text{Type}$ \hspace{1cm} Kinds for proofs and programs
- \text{string} | \text{prin} \hspace{1cm} Base types
- $t \text{ says } t$ \hspace{1cm} Says modality
- $\text{pf } t$ \hspace{1cm} Proof modality
- $(x:t) \rightarrow t$ \hspace{1cm} Logical implication/function arrow/logical quantification/polymorphism

- **pf** monad stops computational effects from polluting the logic.
- **says** modality represents affirmation.
- **Dependent $\rightarrow$** makes for expressive propositions.
Aura’s \textit{says} modality represents affirmation.

- The proposition “principal Alice affirms proposition $P$.”

  $$\text{Alice } \text{says } P : \text{Prop}$$

- Principals may actively affirm propositions with signatures.

  $$\text{sign}(\text{Alice}, P) : \text{Alice } \text{says } P$$

- Principals affirm propositions for which there is evidence.

  $$\text{return}_s \text{Alice } p : \text{Alice } \text{says } P \text{ when } p : P$$
Dependent types allow for expressive rules.

Example (Bob acts for Alice)

Alice says \(((P: \text{Prop}) \rightarrow \text{Bob says } P \rightarrow P)\)
Dependent types allow for expressive rules.

Example (Bob acts for Alice)

Alice \textbf{says} \ ((P : \text{Prop}) \rightarrow \text{Bob says} \ P \rightarrow P)

Example (Bob acts for Alice only regarding validity)

Alice \textbf{says} \ ((x : \text{string}) \rightarrow \text{Bob says} \ \text{valid} x \rightarrow \text{valid} x)
Effect `say` reifies the program’s authority as a signature.

- Programs can manufacture new `sign` objects with `say`.
- Intuitively `say` uses the program's (e.g. current user’s) private key to generate the signature.
- Special principal `self` stands in for the program.

\[
\text{say } P : \text{self says } P
\]

\[
\text{say } P \rightarrow \text{sign}(\text{self}, P)
\]
Effect `say` reifies the program’s authority as a signature.

- Programs can manufacture new `sign` objects with `say`.
- Intuitively `say` uses the program’s (e.g. current user’s) private key to generate the signature.
- Special principal `self` stands in for the program.

\[
say P : pf(self \ says \ P)
\]

\[
say P \rightarrow return_s (sign(self, P))
\]

Technical Point

The `pf` monad is needed to protect props
Aura’s core metatheory formalized in Coq.

- Terms are *locally nameless*, with DeBruijn indexed bound variables and named free variables.
- Formalized features: inductive data types, Prop and Type language fragments, says and pf modalities.
- Main results: Type soundness and decidability of typechecking

**Development Size (in lines of commented Coq code)**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Definitions</td>
<td>1400</td>
</tr>
<tr>
<td>Type Soundness</td>
<td>6000</td>
</tr>
<tr>
<td>Decidability of Typechecking</td>
<td>5000</td>
</tr>
</tbody>
</table>
Aura is (becoming) real.

- **Current Features:**
  - Interpreter and typechecker for full language
  - Foreign function interface
- **Coming Soon:**
  - Cryptographic implementation of `sign`.
  - Automatic logging.
- **Future Research:**
  - Type inference?
  - Surface syntax?
  - Information flow?
  - Effects tracking?
Aura is (becoming) real.

- **Current Features:**
  - Interpreter and typechecker for full language
  - Foreign function interface

- **Coming Soon:**
  - Cryptographic implementation of `sign`.
  - Automatic logging.

- **Future Research:**
  - Type inference?
  - Surface syntax?
  - Information flow?
  - Effects tracking?

Demo
Aura is a framework for proof carrying authorization and audit.

Aura includes

- a small and generic trusted computing base,
- an expressive authorization logic, and
- a principled audit methodology.

Code and papers available from
http://www.cis.upenn.edu/~stevez/sol/aura.html
Aura is a framework for proof carrying authorization and audit.

Aura includes

- a small and generic trusted computing base,
- an expressive authorization logic, and
- a principled audit methodology.

Code and papers available from
http://www.cis.upenn.edu/~stevez/sol/aura.html
• Logging and Proof Reduction
• Typing Rules for Pair and Arrow
• Logical Properties of Aura₀
• Strong Normalization Proof for Aura₀
When something unexpected happens: look at the log.

- System design guarantees a one-to-one correspondence between log entries and resource state changes.
- If a Alice’s signature does not appear in a log entry, she could not have caused the associated action.
- Proofs get convoluted, but proof reduction can restore clarity.

Example (A convoluted proof)

\[(\lambda x.\lambda y.y) (\text{sign}(\text{Alice}, P)) (\text{sign}(\text{Bob}, Q))\]

Here Alice’s signature is “irrelevant.”
Reduction relation includes special cases for **bind**.

### R-Bind-Specious

\[
\begin{align*}
x \notin \text{fv}(t_2) \\
\text{bind}_s x = t_1 \text{ in } t_2 \rightarrow t_2
\end{align*}
\]

Drops unused hypotheses. Most interesting when \( t_1 \) is a signature.

### R-Bind-Commutate

\[
\begin{align*}
y \notin \text{fv}(t_3) \\
\text{bind}_s x = (\text{bind}_s y = t_1 \text{ in } t_2) \text{ in } t_3 \rightarrow \\
\text{bind}_s y = t_1 \text{ in } \text{bind}_s x = t_2 \text{ in } t_3
\end{align*}
\]

Commutation rule that can enable further reductions.

\[\ldots\text{plus standard } \beta \text{ and structural rules.}\]
Proof reduction

Example (The same convoluted proof)

\[(\lambda x. \lambda y. y) (\text{sign}(\text{Alice}, P)) (\text{sign}(\text{Bob}, Q)) \rightarrow^\ast \text{sign}(\text{Bob}, Q)\]

Example (Reducing by special \textit{bind} rules)

\[
\begin{align*}
\text{bind}_s x &= (\text{bind}_s y = (f \text{sign}(\text{Bob}, Q)) \\
&\text{in } \text{sign}(\text{Alice}, P)) \\
&\text{in } x \\
&\rightarrow^\ast \text{sign}(\text{Alice}, P)
\end{align*}
\]
Typing rules for arrow and pair.

\[
\begin{align*}
\Sigma; \Gamma & \vdash t_1 : k_1 \quad \Sigma; \Gamma, x : t_1 & \vdash t_2 : k_2 \\
k_1 \in \{\text{Kind}^P, \text{Prop}, \text{Type}\} & \quad k_2 \in \{\text{Prop}, \text{Type}\} \\
\Sigma; \Gamma & \vdash (x : t_1) \rightarrow t_2 : k_2
\end{align*}
\]

\[
\begin{align*}
\Sigma; \Gamma & \vdash t_1 : k_1 \quad \Sigma; \Gamma, x : t_1 & \vdash t_2 : k_2 \\
k_1, k_2 \in \{\text{Prop}, \text{Type}\} & \\
\Sigma; \Gamma & \vdash \{x : t_1 ; t_2\} : k_1
\end{align*}
\]
Theorem (Confluence)

If $p \rightarrow^* p_1$, and $p \rightarrow^* p_2$, then there exists $p_3$ such that $p_1 \rightarrow^* p_3$ and $p_2 \rightarrow^* p_3$.

Theorem (Strong Normalization)

If $\Gamma \vdash p : s$, then $p$ is strongly normalizing. That is, all reduction sequences starting with $p$ halt.
Aura_0 is strongly normalizing.

Fact

The Calculus of Constructions with dependent pairs is SN.

[Geuvers 95]

Proof Idea

Show Aura reductions can be simulated in a terminating system based on Constructions.

\[
p_0 \rightarrow \rightarrow p_1 \rightarrow \rightarrow p_2 \rightarrow \cdots
\]

\[
t_0 \rightarrow cc t_1 \rightarrow cc t'_1 \rightarrow cc t_2 \rightarrow cc \cdots
\]
Translation from Aura to CoC erases \textbf{says}

\begin{definition}
\begin{align*}
[A \ \text{says} \ P] & \approx [P] \\
[\text{return}_s A \ p] & \approx [p] \\
[\text{bind}_s x = p: P \ \text{in} \ q] & \approx (\lambda x:[P].[q]) \ [p] \\
[\text{sign}(A, P)] & \approx x \ \text{fresh}
\end{align*}
\end{definition}

Where $\Delta$ maps signatures to unique fresh variables.

(We'll treat the $\Delta$'s implicitly from now on.)

Lemma
If $p$ is a well typed term in Aura $\text{0}$, then—for an appropriate context—$[p]$ is well typed in CoC.
Translation from Aura to CoC erases \textit{says}

### Definition

\[
[A \ \text{says} \ P]_\Delta \ \approx \ \ [P]_\Delta
\]

\[
[\text{return}_s \ A \ p]_\Delta \ \approx \ \ [p]_\Delta
\]

\[
[\text{bind}_s \ x = p: P \ \text{in} \ q]_\Delta \ \approx \ \ (\lambda x:[P]_\Delta. [q]_\Delta) \ [p]_\Delta
\]

\[
[\text{sign}(A, P)]_\Delta \ \approx \ \Delta(\text{sign}(A, P))
\]

Where $\Delta$ maps signatures to unique fresh variables. (We’ll treat the $\Delta$’s implicitly from now on.)
Translation from Aura to CoC erases `says`

**Definition**

\[
\begin{align*}
[A \ says \ P]_\Delta & \approx [P]_\Delta \\
[\text{return}_s \ A \ p]_\Delta & \approx [p]_\Delta \\
[\text{bind}_s \ x = p: P \ in \ q]_\Delta & \approx (\lambda x:[P]_\Delta \cdot [q]_\Delta) \ [p]_\Delta \\
[\text{sign}(A, P)]_\Delta & \approx \Delta(\text{sign}(A, P))
\end{align*}
\]

Where \( \Delta \) maps signatures to unique fresh variables. (We’ll treat the \( \Delta \)’s implicitly from now on.)

**Lemma**

*If \( p \) is a well typed term in Aura\(_0\), then—for an appropriate context—\([p] \ is well typed in CoC.*
Constructions needs a new reduction to simulate \textbf{bind}.

**Definition (CC' reduction)**

The CC' reduction relation augments the standard Calculus of Construction reduction relation with

\[
(\lambda x: t_1)((\lambda y: s \ t_2)u) \rightarrow (\lambda y: s. ((\lambda x: t_1) t_2))u
\]

\(\beta'\) reductions simulate R-Bind-Commute reductions:

\[
y \notin \text{fv}(t_3)
\]

\[
\text{bind}_s x = (\text{bind}_s y = t_1 \text{ in } t_2) \text{ in } t_3 \rightarrow \\
\text{bind}_s y = t_1 \text{ in } \text{bind}_s x = t_2 \text{ in } t_3
\]
Aura_0 is strongly normalizing.

Lemma

Well typed CoC terms are SN under CC’ reduction.

[Lindley 05]

Lemma

If \( p \) is a well typed Aura_0 term and \( p \rightarrow p' \). Then
\[
[p] \longrightarrow^{+}_{CC'} [p']
\] using one or more steps.

Proof of strong normalization.

Imagine \( p \) is an looping Aura_0 term. Then by the second lemma we can build a term which loops according to CC’. This contradicts the first lemma.