AuraConf: A Unified Approach to Authorization and Confidentiality

Jeff Vaughan

Department of Computer Science
University of California, Los Angeles

TLDI
January 25, 2011
Some attackers don’t play fair.

\[
\text{playFor: } (s: \text{Song}) \rightarrow (p: \text{prin}) \rightarrow \text{pf} \ (\text{RecCo says} \ (\text{MayPlay} \ p \ s)) \rightarrow \text{Mp3Of} \ s
\]
Some attackers don’t play fair.

playFor: (s: Song) → (p: prin) →
    pf (RecCo says (MayPlay p s)) → Mp3Of s
Some attackers don’t play fair.

```
playFor: (s: Song) → (p: prin) →
    pf (RecCo says (MayPlay p s)) → Mp3Of s
```
AURA_{conf} protects confidential data.

- Types provide a formal description of confidentiality policy.
AURA\textsubscript{conf} protects confidential data.

- Types provide a formal description of confidentiality policy.
- Encryption provides an enforcement mechanism.
\textbf{AURA}_{\text{conf}}$ protects confidential data.

- Types provide a formal description of confidentiality policy.
- Encryption provides an enforcement mechanism.
- \textit{Blame} mechanism allows audit of (some) failures.
First thought: borrow someone else’s idea!

- Direct use of cryptography
  - Applied Crytpo. [Schneier ’96]

- Language operations supporting cryptography
  - Spi Calculus [Abadi+ ’98], $\lambda_{seal}$ [Sumii+ ’04]

- Type-based information flow
  - Aura [Jia & Zdancewic ’09]

- Information flow + explicit cryptography
  - Key-Based DLM [Chothia+ ’03], [Askarov+ ’06]

- Declarative policy enforcement by automatic encryption
  - SImp [Vaughan & Zdancewic ’06]
First thought: borrow someone else’s idea!

- Direct use of cryptography
  - Applied Crypto. [Schneier ’96]

- Language operations supporting cryptography
  - Spi Calculus [Abadi+ ’98], $\lambda_{\text{seal}}$ [Sumii+ ’04]

- Type-based information flow
  - Aura [Jia & Zdancewic ’09]

- Information flow + explicit cryptography
  - Key-Based DLM [Chothia+ ’03], [Askarov+ ’06]

- Declarative policy enforcement by automatic encryption
  - SImp [Vaughan & Zdancewic ’06]

None of these are good fits with AURA.
New mechanism, for types describe encrypted data.

\[
\text{playForEnc}: (s: \text{Song}) \rightarrow (p: \text{prin}) \rightarrow \\
\quad \text{pf} \ (\text{RecCo says MayPlay p s}) \rightarrow \\
\quad (\text{Mp3Of s}) \text{ for p}
\]
New mechanism, **for** types describe encrypted data.

\[
\text{playForEnc: (s: Song) } \rightarrow \ (p: \text{ prin}) \rightarrow \ \text{pf (RecCo says MayPlay p s)} \rightarrow \ (\text{Mp3Of s) for p}}
\]
New mechanism, for types describe encrypted data.

```
playForEnc: (s: Song) → (p: prin) →
  pf (RecCo says MayPlay p s) →
  (Mp3Of s) for p
```
New mechanism, for types describe encrypted data.

\[
\text{playForEnc: } (s: \text{Song}) \rightarrow (p: \text{prin}) \rightarrow \\
\quad \text{pf} \ (\text{RecCo says MayPlay p s}) \rightarrow \\
\quad \text{(Mp3Of s) for p}
\]
Outline

1. Introduction
2. Overview of for types
3. Feature design
4. Language theory
5. Conclusion
Overview of for types
\( \textit{AURA}_{\text{conf}} \) represents confidentiality monadically: return.

\[
\text{return} \ Alice \ 42: \ \text{int} \ \textbf{for} \ Alice
\]

N.B.

Monads are a common Haskell design pattern:

- **return**: creates an object
- **run**: consumes an object
- **bind**: composes objects
AURA_{conf} represents confidentiality monadically: return.

\[ \text{return Alice 42: int for Alice} \]

\[ \mathcal{E}(\text{Alice, 42, 0x32A3}) \]
and some metadata

N.B.

Monads are a common Haskell design pattern:
- **return**: creates an object
- **run**: consumes an object
- **bind**: composes objects
AURA\textsubscript{conf} represents confidentiality monadically: run.

\[
\text{run (return Alice 42): int}
\]
\textsc{AURA}_{\text{conf}} \text{ represents confidentiality monadically: run.}

\begin{equation*}
\begin{aligned}
\text{run} \ (\text{return} \text{ Alice} \ 42): \text{ int} \\
\end{aligned}
\end{equation*}

\geq

\begin{equation*}
\begin{aligned}
42
\end{aligned}
\end{equation*}
AURA\textsubscript{conf} represents confidentiality monadically: run.

\[
\text{run (return Alice 42): int} \\
\downarrow \\
42
\]

- **run** can fail on “bad” ciphertext.
  - wrong decryption key
  - ill-formed/ill-typed payload plaintext
  - corrupt ciphertext

- **run** \( e \leadsto e' \) where \( e' \) blames \( p \).
$\textbf{bind} \ (\text{int} \ \text{for} \ \text{Alice})$

$(\text{return} \ \text{Alice} \ 21)$

$(\lambda \{ \_ \} \ x: \text{int}. \ \text{return} \ \text{Alice} \ (2^*x))$

: $\text{int} \ \text{for} \ \text{Alice}$
$\text{bind}$ (int for Alice)
  (return Alice 21)
  ($\lambda\{\_\} x: \text{int}. \text{return} \text{Alice} \ (2 \times x)$)
  : int for Alice

$\Downarrow$

$\mathcal{E}(\text{Alice},$
  ($\lambda\{\_\} x: \text{int}. \text{return} \ 2 \times x$) (run $\mathcal{E}(\text{Alice}, \ 21, \ 0x32A4)$)
  $0x32A3)$

and some metadata
\textsc{AURA}_{\text{conf}} \text{ represents confidentiality monadically: bind.}

\[
\text{bind} \ (\text{int\ for\ Alice}) \\
\qquad (\text{return\ Alice\ 21}) \\
\qquad (\lambda \{\_\} \ x: \text{int}. \ \text{return\ Alice\ (2\times x))} \\
\qquad : \text{int\ for\ Alice}
\]

\[
\Downarrow
\]

\[\mathcal{E}(\text{Alice,} \\
\qquad (\lambda \{\_\} \ x: \text{int}. \ \text{return\ 2\times x}) \ (\text{run } \mathcal{E}(\text{Alice, 21, 0x32A4})) \\
\qquad 0x32A3) \\
\]

and some metadata

\[\approx \mathcal{E}(\text{Alice, 42, 0x32A5}) \\
\quad \text{and some metadata}\]
\( \text{bind} \ (\text{int} \ \text{for} \ \text{Alice}) \)
\( (\text{return} \ \text{Alice} \ 21) \)
\( (\lambda \{ \_ \} \ x: \ \text{int}. \ \text{return} \ \text{Alice} \ (2 \times x)) \)
\( : \ \text{int} \ \text{for} \ \text{Alice} \)

\( \Downarrow \)

\( \mathcal{E}(\text{Alice}, \)
\( (\lambda \{ \_ \} \ x: \ \text{int}. \ \text{return} \ 2 \times x) \ (\text{run} \ \mathcal{E}(\text{Alice}, \ 21, \ 0x32A4)) \)
\( 0x32A3) \)

and some metadata

\( \approx \mathcal{E}(\text{Alice}, \ 42, \ 0x32A5) \)
and some metadata

This is mobile code
Programs may dynamically load data or code with **run**
- Dynamic type-checking needed to catch errors
- Ciphertexts may be paired with digitally signed proofs describing their contents
- In case of emergency, evaluation “blames” such proofs

Well-typed clients create values that don’t cause blame
- Typing of **bind** makes sure mobile expressions can be correctly decrypted by the receiver
- Receiver’s dynamic resources are modeled by sender’s typechecker
Feature design
The tension in $\text{AURA}_{\text{conf}}$’s design.

Suppose expression $e$ contains secrets. A client analyzing $e$ is:
The tension in $\text{AURA}_{\text{conf}}$’s design.

Suppose expression $e$ contains secrets. A client analyzing $e$ is:

Good!

Type Theorist
The tension in $\text{AURA}_{\text{conf}}$’s design.

Suppose expression $e$ contains secrets. A client analyzing $e$ is:

- Type Theorist: **Good!**
- Cryptographer: **Bad!**
Challenge 1: Typing is relative.

\textbf{return} Alice "toaster"
\( \varepsilon(Alice, "toaster", 0x0312) \)
$\varepsilon(Alice, "toaster", 0x0312)$

I can typecheck this because I know its provenance.

Bob
\text{Challenge 1: Typing is relative.}

\varepsilon(Alice, "toaster", 0x0312)
$\texttt{(Alice, "toaster", 0x0312)}$
\( \mathcal{E}(\text{Alice, "toaster", 0x0312}) \)

I can typecheck this with my private key.

Alice

Bob
Challenge 1: Typing is relative.

\(\varepsilon(Alice, \text{"toaster"}, 0x0312)\)

Alice

Bob
Challenge 1: Typing is relative.

\[ E(Alice, \text{"toaster"}, 0x0312) \]
\text{Challenge 1: Typing is relative.}

\( \mathcal{E}(\text{Alice, "toaster", 0x0312}) \)

The ciphertext looks like noise to me.

Alice
Bob
Charlie
Challenge 1: Typing is relative.

\( \varepsilon(Alice, \text{"toaster"}, 0x0312) \)
True cast

$$\text{cast } \mathcal{E}(a, e, n) \to \text{ (int for Alice)} : \text{ int for Alice}$$

- Possible if typechecker can statically decrypt $\mathcal{E}(a,e,n)$.
- Also possible if the typechecker has a prerecorded fact, attesting to the form of $\mathcal{E}(a,e,n)$. 
Metadata casts guide typing of ciphertexts.

**True cast**

<table>
<thead>
<tr>
<th>cast $E(a, e, n)$ to (int for Alice): int for Alice</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Possible if typechecker can statically decrypt $E(a,e,n)$.</td>
</tr>
<tr>
<td>• Also possible if the typechecker has a prerecorded fact, attesting to the form of $E(a,e,n)$.</td>
</tr>
</tbody>
</table>

**Justified cast**

<table>
<thead>
<tr>
<th>cast $E(a, e, n)$ to (int for Alice) blaming p: int for Alice</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Valid when p: c says ($E(a,e,n)$ isa (int for Alice)).</td>
</tr>
<tr>
<td>• Proof p can be blamed for decryption or typing failures.</td>
</tr>
</tbody>
</table>
Decryption failures may be audited with justified casts.
Decryption failures may be audited with justified casts.
Decryption failures may be audited with justified casts.

Evidence: mentions Mal
Action: blame Mal
Evidence: ill-formed
Action: ignore message
Evidence: mentions Alice
Action: blame Alice

Evidence: ill-formed
Action: ignore message
Decryption failures may be audited with justified casts.
Decryption failures may be audited with justified casts.

Evidence: mentions Mal
Action: blame Mal
Evidence: ill-formed
Action: ignore message
Evidence: mentions Alice
Action: blame Alice
Decryption failures may be audited with justified casts.

Evidence: mentions Mal
Action: blame Mal
Evidence: ill-formed
Action: ignore message
Evidence: mentions Alice
Action: blame Alice
Decryption failures may be audited with justified casts.

Evidence: mentions Mal
Action: blame Mal
Evidence: ill-formed
Action: ignore message
Evidence: mentions Alice
Action: blame Alice

Evidence: mentions Mal
Action: blame Mal
Evidence: mentions Alice
Action: blame Alice
Dynamic semantics
- Keys are required at runtime to implement run (and say).
- Type-and-effect analysis tracks these keys.
- FX [Lucassen+ ’88], foundations [Talpin+ ’92]

Static semantics
- True casts need keys at compile time for typechecking.
- Tracked using ideas from modal type systems.
- Modal Proofs as Distributed Programs [Jia+ 04], ML5 [Murphy ’08]

Combining these analyses is interesting!
Challenge 3: Typing exhibits history-dependence.

Consider Bob preparing a confidential message for Alice

```
return Alice 3  ⇝  cast $\epsilon(-)$ to int for Alice
```

Naively: Bob lacks Alice’s private key—he can’t typecheck this.

Solution

Evaluation semantics creates new facts to guide the typechecker.

- This ensures types are preserved at runtime and programs don’t “go wrong.”
Language theory
Evaluation tracks fact generation and authority.

\[ \Sigma; \mathcal{F}_0; W \vdash \{e, n\} \rightarrow \{e', n'\} \text{ learning } \mathcal{F} \]
Evaluation tracks fact generation and authority.

\[ \Sigma; \mathcal{F}_0; W \vdash \{|e, n|\} \rightarrow \{|e', n'|\} \text{ learning } \mathcal{F} \]

- *e* steps to *e'*. 

Randomization seed *n* is updated to *n'*. 

Key *W* is available for signing and decrypting.

"The program is running with *W*'s authority."

Signature \( \Sigma \), facts context \( \mathcal{F}_0 \), and key \( W \) are available for dynamic type-checking.

New facts \( \mathcal{F} \) are produced during encryptions.
Evaluation tracks fact generation and authority.

\[ \Sigma; \mathcal{F}_0; W \vdash \{e, n\} \rightarrow \{e', n'\} \text{ learning } \mathcal{F} \]

- e steps to \(e'\).
- Randomization seed \(n\) is updated to \(n'\).
Evaluation tracks fact generation and authority.

\[\Sigma; \mathcal{F}_0; W \vdash \{e, n\} \rightarrow \{e', n'\}\] learning \(\mathcal{F}\)

- \(e\) steps to \(e'\).
- Randomization seed \(n\) is updated to \(n'\).
- Key \(W\) is available for signing and decrypting.
  “The program is running with \(W\)’s authority.”
Evaluation tracks fact generation and authority.

\[ \Sigma; F_0; W \vdash \{ e, n \} \rightarrow \{ e', n' \} \text{ learning } F \]

- e steps to e'.
- Randomization seed n is updated to n'.
- Key W is available for signing and decrypting.
  “The program is running with W’s authority.”
- Signature Σ, facts context F_0, and key W are available for dynamic type-checking.
Evaluation tracks fact generation and authority.

\( \Sigma; \mathcal{F}_0; W \vdash \{ |e, n| \} \rightarrow \{ |e', n'| \} \) learning \( \mathcal{F} \)

- e steps to \( e' \).
- Randomization seed \( n \) is updated to \( n' \).
- Key \( W \) is available for signing and decrypting.
  “The program is running with \( W \)'s authority.”
- Signature \( \Sigma \), facts context \( \mathcal{F}_0 \), and key \( W \) are available for dynamic type-checking.
- New facts \( \mathcal{F} \) are produced during encryptions.
Evaluation tracks fact generation and authority.

\[ \Sigma; \mathcal{F}_0; W \vdash \{e, n\} \rightarrow \{e', n'\} \text{ learning } \mathcal{F} \]

- e steps to e'.
- Randomization seed n is updated to n'.
- Key W is available for signing and decrypting. “The program is running with W’s authority.”
- Signature \( \Sigma \), facts context \( \mathcal{F}_0 \), and key W are available for dynamic type-checking.
- New facts \( \mathcal{F} \) are produced during encryptions.
Anatomy of the typing relation.

\[ \Sigma; \mathcal{F}; W; \Gamma; U; V \vdash e : t \]
Anatomy of the typing relation.

\[ \Sigma; \mathcal{F}; W; \Gamma; U; V \vdash e : t \]

- \( e \) has type \( t \) w.r.t. \( \Gamma \)'s free variables and \( \Sigma \)'s type definitions.
Anatomy of the typing relation.

\[ \Sigma; \mathcal{F}; W; \Gamma; U; V \vdash e : t \]

- \( e \) has type \( t \) w.r.t. \( \Gamma \)'s free variables and \( \Sigma \)'s type definitions.
- Facts in \( \mathcal{F} \) summarize knowledge about ciphertexts.
Anatomy of the typing relation.

\[ \Sigma; \mathcal{F}; W; \Gamma; U; V \vdash e : t \]

- \( e \) has type \( t \) w.r.t. \( \Gamma \)'s free variables and \( \Sigma \)'s type definitions.
- Facts in \( \mathcal{F} \) summarize knowledge about ciphertexts.
- *Statically available key* \( W \) indicates keys available for typechecking.
Anatomy of the typing relation.

\[ \Sigma; \mathcal{F}; W; \Gamma; U; V \vdash e : t \]

- \( e \) has type \( t \) w.r.t. \( \Gamma \)'s free variables and \( \Sigma \)'s type definitions.
- Facts in \( \mathcal{F} \) summarize knowledge about ciphertexts.
- **Statically available key** \( W \) indicates keys available for typechecking.
- **Soft decryption limit** \( U \) specifies a subset of \( W \) safe to use currently.
Anatomy of the typing relation.

\[ \Sigma; \mathcal{F}; W; \Gamma; U; V \vdash e : t \]

- \( e \) has type \( t \) w.r.t. \( \Gamma \)'s free variables and \( \Sigma \)'s type definitions.
- Facts in \( \mathcal{F} \) summarize knowledge about ciphertexts.
- **Statically available key** \( W \) indicates keys available for typechecking.
- **Soft decryption limit** \( U \) specifies a subset of \( W \) safe to use currently.
- **Effects label** \( V \) summarizes the keys needed to run \( e \).
Anatomy of the typing relation.

\[ \Sigma; \mathcal{F}; W; \Gamma; U; V \vdash e : t \]

- e has type t w.r.t. \( \Gamma \)'s free variables and \( \Sigma \)'s type definitions.
- Facts in \( \mathcal{F} \) summarize knowledge about ciphertexts.
- **Statically available key** \( W \) indicates keys available for typechecking.
- **Soft decryption limit** \( U \) specifies a subset of \( W \) safe to use currently.
- **Effects label** \( V \) summarizes the keys needed to run \( e \).
Anatomy of the typing relation.

\[ \Sigma; \mathcal{F}; W; \Gamma; U; V \vdash e : t \]

- \( e \) has type \( t \) w.r.t. \( \Gamma \)'s free variables and \( \Sigma \)'s type definitions.
- Facts in \( \mathcal{F} \) summarize knowledge about ciphertexts.
- **Statically available key** \( W \) indicates keys available for typechecking.
- **Soft decryption limit** \( U \) specifies a subset of \( W \) safe to use currently.
- **Effects label** \( V \) summarizes the keys needed to run \( e \).

soft decryption limit \( \sim \) modal-logic world

effects label \( \sim \) standard type-and-effects label
Soundness requires handling fact contexts explicitly.

**Definition (valid$_\Sigma$ $\mathcal{F}$)**

valid$_\Sigma$ $\mathcal{F}$ holds when

1. $\Sigma$ is well formed: $\Sigma \vdash \Diamond$.
2. Facts are true: $\mathcal{E}(a, e, n) : t$ for $b \in \mathcal{F}$ implies $a = b$ and $\Sigma; \cdot; b; \cdot; b; b \vdash e : t$. 

**Lemma (New Fact Validity)**

Assume valid$_\Sigma$ $\mathcal{F}$ and $\Sigma; F_0; W; \Gamma; U; V \vdash e : t$. Then valid$_{\Sigma; F_0; W} \vdash \{ |e, n| \} \rightarrow \{ |e', n'| \}$ learning $\mathcal{F}$ implies valid$_\Sigma$ $\mathcal{F}$.

**Slogan**

Preservation + Progress + New Fact Validity = Soundness

Soundness results mechanized in Coq.
Soundness requires handling fact contexts explicitly.

**Definition (valid$_\Sigma$ $\mathcal{F}$)**

valid$_\Sigma$ $\mathcal{F}$ holds when

1. $\Sigma$ is well formed: $\Sigma \vdash \Diamond$.
2. Facts are true: $\mathcal{E}(a, e, n) : t$ for $b \in \mathcal{F}$ implies $a = b$ and $\Sigma; \cdot; b; \cdot; b; b \vdash e : t$.

**Lemma (New Fact Validity)**

Assume valid$_\Sigma$ $\mathcal{F}_0$ and $\Sigma; \mathcal{F}_0; W; \Gamma; U; V \vdash e : t$. Then $\Sigma; \mathcal{F}_0; W \vdash \{[e, n]\} \to \{[e', n']\}$ learning $\mathcal{F}$ implies valid$_\Sigma$ $\mathcal{F}$.
Soundness requires handling fact contexts explicitly.

**Definition (valid_Σ ℱ)**

valid_Σ ℱ holds when

1. Σ is well formed: Σ ⊢ ⊤.
2. Facts are true: \( \mathcal{E}(a, e, n) : t \) for \( b \in ℱ \) implies \( a = b \) and \( Σ; \cdot; b; \cdot; b; b ⊢ e : t \).

**Lemma (New Fact Validity)**

Assume valid_Σ ℱ₀ and \( Σ; ℱ₀; W; Γ; U; V ⊢ e : t \). Then \( Σ; ℱ₀; W ⊢ \{ |e, n| \} \rightarrow \{ |e', n'| \} \) learning ℱ implies valid_Σ ℱ.

**Slogan**

Preservation + Progress + New Fact Validity = Soundness
Soundness requires handling fact contexts explicitly.

**Definition (valid$_\Sigma$ $\mathcal{F}$)**

valid$_\Sigma$ $\mathcal{F}$ holds when

1. $\Sigma$ is well formed: $\Sigma \vdash \Diamond$.
2. Facts are true: $e(a, e, n) : t$ for $b \in \mathcal{F}$ implies $a = b$ and $\Sigma; \cdot; b; \cdot; b; b \vdash e : t$.

**Lemma (New Fact Validity)**

Assume valid$_\Sigma$ $\mathcal{F}_0$ and $\Sigma; \mathcal{F}_0; W; \Gamma; U; V \vdash e : t$. Then $\Sigma; \mathcal{F}_0; W \vdash \{|e, n|\} \rightarrow \{|e', n'|\}$ learning $\mathcal{F}$ implies valid$_\Sigma$ $\mathcal{F}$.

**Slogan**

Preservation + Progress + New Fact Validity = Soundness
Noninterference: Secrets don’t affect public outputs.

\[ b \vdash \text{Aura Program} \]
Noninterference: Secrets don’t affect public outputs.

\[ \mathcal{E}(Alice, \text{"toaster"}, 0x0399) \]

: string for Alice

b \leftarrow Aura Program
Noninterference: Secrets don’t affect public outputs.

\[ \mathcal{E}(\text{Alice, "toaster", 0x0399}) \]

: string for Alice

\[ b \leftarrow \text{Aura Program} \]
Noninterference: Secrets don’t affect public outputs.

\[ \text{Aura Program} \]
\( (\text{Alice, "toaster", 0x0399}) \)

: string for Alice

\[ b \vdash \]

Aura Program

15
Noninterference: Secrets don’t affect public outputs.

\(\mathcal{E}(\text{Alice, } "\text{toaster}", \; 0x0399)\):

: string for Alice

b \leftarrow Aura Program
Noninterference: Secrets don’t affect public outputs.

\( (\mathcal{E}(\text{Alice, "lambda", 0x0312}) \vdash \text{string for Alice} \)

b \leftarrow \text{Aura Program}
Noninterference: Secrets don’t affect public outputs.

\[ \mathcal{E}(\text{Alice, "lambda", 0x0312}) \]

: string \textbf{for} Alice

b \leftarrow \text{Aura Program}
Noninterference: Secrets don’t affect public outputs.

\[ \mathcal{E}(\text{Alice, "lambda", 0x0312}) \]

: string for Alice

\[ b \vdash \text{Aura Program} \]

15
Noninterference: Secrets don’t affect public outputs.

\[ E(Alice, "\text{"lambda"}", 0x0312) \]

: string for Alice

\[ b \vdash \text{Aura Program} \]

15
Conclusion
Summary

- Type specification + cryptographic enforcement
  $\mapsto$ confidentiality

- Type-and-effects analysis + modal-type theory
  $\mapsto$ precise resource tracking

- $\text{AURA}_{\text{conf}}$ unifies mechanisms for confidentiality, audit and access control.
Acknowledgments

Thank you to all my collaborators on Aura project!

- Limin Jia
- Karl Mazurak
- Joseph Schorr
- Luke Zarko
- Steve Zdancewic
- Jianzhou Zhao
Acknowledgments

Thank you to all my collaborators on Aura project!

- Limin Jia
- Karl Mazurak
- Joseph Schorr
- Luke Zarko
- Steve Zdancewic
- Jianzhou Zhao

Questions?