Combining
Access Control and Information Flow
in DCC
(work in progress)

Steve Zdancewic
University of Pennsylvania

In collaboration with: Martín Abadi,
Karl Mazurak and Jeff Vaughan
Dependency Core Calculus (DCC)

- **A Core Calculus of Dependency**
  [Abadi, Banerjee, Heintz, Riecke: POPL 1999]
  - Monadic type system with lattice of "labels" $T_L$
  - Key property: noninterference
  - Showed how to encode many dependency analyses: information flow, binding time analysis, slicing, etc.

- **Access control in a Core Calculus of Dependency**
  [Abadi: ICFP 2006]
  - Essentially the *same* type system is an authorization logic
  - Instead of $T_L$ read the type as "L says T"
  - Curry-Howard isomorphism "programs are proofs"

- **Question:** *Can these two different interpretations be combined in a sensible way?*
Goal of this work:

• Develop a programming language that exploits these two interpretations of DCC:
  – Proof-carrying Authorization
    [Appel & Felton 1999] [Bauer et al. 2002]
  – Strong information-flow properties
    (as in Jif [Myers et al.] , FlowCaml [Pottier & Simonet])

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

• In this talk: A high-level tour of DCC and some of my current thoughts about structuring such a programming language
Polymorphic DCC

• Types // Authorization Logic
  \[ T ::= \text{true} \]
  \[ c \]
  \[ \alpha \]
  \[ T \land T \]
  \[ T \lor T \]
  \[ T \rightarrow T \]
  \[ \forall \alpha.T \]
  \[ P \text{ says } T \]

• Labels // Principals
  \[ P,Q,R,S,\ldots \]

• Ordering:
  \[ P \leq Q \]
  – Labels: "Data labeled with Q is more restricted than data labeled with P"
    \[ \text{untainted} \leq \text{tainted} \]
    or
    \[ \text{public} \leq \text{secret} \]
  – Principals: "P acts for Q" or "P is more trusted than Q"
DCC = Polymorphic \( \lambda \) Calculus +

\[
\begin{align*}
\Gamma |- e : T \\
\hline
\Gamma |- \eta_P e : P \text{ says } T
\end{align*}
\]

\[
\begin{align*}
\Gamma |- e_1 : P \text{ says } T_1 \\
\Gamma, x : T_1 |- e_2 : Q \text{ says } T_2 \\
\Gamma |- P \leq Q \\
\hline
\Gamma |- \text{bind } x = e_1 \text{ in } e_2 : Q \text{ says } T_2
\end{align*}
\]
Authorization Logic Example Theorems

- \( T \rightarrow P \text{ says } T \)
  "Principals assert all true statements"

- \( (P \text{ says } T) \rightarrow (P \text{ says } (T \rightarrow U)) \rightarrow (P \text{ says } U) \)
  "Principals' assertions are closed under deduction"

- If \( P \leq Q \) then \( (P \text{ says } T) \rightarrow (Q \text{ says } T) \)
  "If \( P \) acts for \( Q \) then whatever \( P \) says, \( Q \) says"

- Define "\( P \text{ speaks-for } Q \)" = \( \forall \alpha. (P \text{ says } \alpha) \rightarrow (Q \text{ says } \alpha) \)
- \( (Q \text{ says } (P \text{ speaks-for } Q)) \rightarrow (P \text{ speaks-for } Q) \)
  "\( Q \) can delegate its authority to \( P \)"   (The "hand off" axiom)
Example Non-theorems

- It is not possible to prove false: $\forall T. T$
  - "The logic is consistent"

- It is not possible to prove: $P$ says false
  - "Principals are consistent"

- It is not possible to prove: $\forall T.(A$ says $T) \rightarrow T$
  - "Just because $A$ says it doesn't mean it's true"

- If $\neg (Q \leq P)$ then there is no $T$ such that:
  $(Q$ says $T) \rightarrow P$ says false
  - "Nothing $Q$ can say can cause $P$ to be inconsistent"
Example: File System authorization policy

- P1: FS says Owns(A,F1)
- P2: FS says Owns(B,F2)
  ...
- OwnerControlsRead:
  \[ \forall P,Q,F. \ (FS \ says \ Owns(P,F)) \rightarrow (P \ says \ MayRead(Q,F)) \rightarrow MayRead(Q,F) \]

- Read operation: expects a proof that \textit{MayRead}(A,F1) whenever A requests to read F1
  - [Question: isn't this too static?]
There is no proof of:

$$\forall T. \forall S. \text{Q says } (T \lor S) \rightarrow (\text{Q says } T) \lor (\text{Q says } S)$$

Crucial point: says *doesn't distribute over disjunction*

Authorization Logic:

- *The type above would allow an adversary to control which statement is made by Q.*

Explicit information flow vs. Implicit information flow:

- Explicit = Data (tag on the sum type)
- Implicit = Control (branch taken when destructing the sum)
Noninterference in DCC

• Assume:
  – \( \neg (P \leq Q) \)
  – \( x:(P \text{ says } T) \vdash e : (Q \text{ says } \text{bool}) \)
  – \( \vdash e_1, e_2 : P \text{ says } T \)

• Then:
  \[
e\{e_1/x\} \rightarrow^* v \quad \text{iff} \quad e\{e_2/x\} \rightarrow^* v
  \]

• Corollary: Any term of type
  \( (\text{Tainted says } T) \rightarrow (\text{Untainted says } \text{bool}) \)
  is a constant function.
Summary So Far

• DCC as an information-flow type system:
  – Types express information-flow constraints
  – Well-typed terms are programs that satisfy the information-flow constraints.

• DCC as an authorization logic:
  – Types express authorization policies
  – Well-typed terms are constructive proofs that are evidence of authorization.

• Just use DCC and we're done combining access control and information flow, right?
  – Not quite!
Decentralized Authorization

• Authorization policies require uninterpreted constants or free variables (uninhabited types):
  – e.g. "MayRead(B,F)" or "Owns(A,F)"
  – 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 / distributed implementation:
  – One possible proof that "A says T" is A's digital signature on a string "T", written \( \text{sign}(A, "T") \)
Adding "Say"

• How to create the value $\text{sign}(A, "T")$?
• Requires access to A's private key…
  – Programs run with some "authority" = a private key
  – With A's authority:
    \[
    \text{say("T") evaluates to } \text{sign}(A, "T")
    \]
• What T's should a program be able to say?
  – T's from a statically predetermined set (static auditing)
  – T's from a set determined at load time
    • A bit like Java or C#'s privilege models.
• In any case: log the fact that "T" was said by the program
3 Example Proofs of A says MayRead(B,F)

- \text{sign}(A, "MayRead(B,F)"")
  - Direct authorization via signature

- bind x = \text{sign}(C,"MayRead(B,F)"") in \eta_A x
  - Implicit delegation (assuming C \leq A)

- bind x = \text{sign}(A, "B speaks-for A") in
  x [MayRead(B,F)] sign(B,"MayRead(B,F))
  - Explicit delegation to Q via speaks-for
Auditing programs

• What does the program do with the proofs?
• More Logging!
  – They record justifications of why certain operations were permitted.
• When do you do the logging?
  – Answer: As close to the use of the privileges as possible.
  – Easy for built-in security-relevant operations like file I/O.
  – Also provide a "log" operation for programmers to use explicitly.
• Question: what theorem do you prove?
  – Correspondence between security-relevant operations and log entries.
  – Log entries should explain the observed behavior of the program.
• Speculation: A theory of auditing?
A Problem with Information Flow

- These signatures conflict with DCC as a programming language!
  - Evaluation can get stuck at 'bind' operations because there are now two flavors of inhabitants of type "P says int"
    \((\eta_A 3)\) vs. \(\text{sign}(A, "int")\)

- Solution: separate the "proofs" from other kinds of values
  - Many possible designs
  - Current approach: introduce a new type \(Pf T\)
    - \(Pf T\) is the type of proofs of the proposition \(T\)
    - \(Pf\) is another monad.

- This decouples the authorization-logic component from the programming language component
  - Question: Doesn't this suggest that authorization logic & information flow are largely orthogonal?
  - Answer: Yes!
Ramifications of this separation

- There are no elimination forms for $Pf \; T$
  - Such proof values are used only for logging
  - But...any two values of type $Pf \; T$ are equivalent
  - As a consequence, it is safe to treat these values as having "high integrity"

- To ensure progress, $\text{sign}(A,T)$ can only occur under the $Pf$ term constructor:

  $\Gamma;A \; |- \; T :: Prop$

  $\Gamma;A \; |- \; \text{say}(T) : Pf \; (A \; \text{says} \; T)$
Signing Values?

- What about signing values to vouch for their integrity?
  - Introduce (simple) dependent types:
    \[\{x:T; \text{Pf } T(x)\}\] dependent pairs
    \[(x:T) \rightarrow T(x)\] dependent functions
  - (Restrict the dependency domain to first order data.)
    - Alternative: use singleton types
    - Question: best practice for "lightweight dependency"
  - Invariant: sign only types
    - computation can't depend on signatures
    - But, can use predicates: \[\{F:\text{File}; \text{Pf } FS \text{ says Owns}(A,F)\}\]
Example authorization policy (revised)

- **getOwner**: \((F:\text{File}) \rightarrow \text{Maybe} (\exists P. \text{Pf} \text{ FS says Owns}(P,F))\)

- **OCR (OwnerControlsRead)**:
  \[ \forall P,Q. (F:\text{File}) \rightarrow (\text{FS says Owns}(P,F)) \rightarrow (P \text{ says MayRead}(Q,F)) \rightarrow \text{MayRead}(Q,F) \]

- **send**: \(\forall Y. (F:\text{File}) \rightarrow \text{Pf MayRead}(Y,F) \rightarrow \text{true}\)
  - Sends the file F to Y (via side effects)
  - Logs the proof that Y may read F
Implementing a request handler

- Type $\text{Req} = \exists P, Q, \{F: \text{File}, \text{Pf} P \text{ says MayRead}(Q,F)\}$
- $\text{HandleReq} : \text{Req} \rightarrow \text{true} =$

\[
\lambda r: \text{Req}.
\quad \text{let } P, Q, \{F; p\} = r \text{ in}
\quad \text{case (getOwner } f) \text{ of}
\quad \quad \text{Nothing} => ()
\quad | \text{Just } P', q =>
\quad \quad \quad \text{if } P = P' \text{ then}
\quad \quad \quad \quad \text{send } [Q] F (\text{letPf } x = p \text{ in letPf } y = q \text{ in Pf (OCR } [P] [Q] F y x))
\]
Status

• We have a core calculus worked out on paper:
  – DCC + constants + sign
    • for access control
  – DCC + Pf + (simple) dependent types
    • for information-flow
  – Another connection declassification: A says t → t

• Still in the process of doing the proofs
  – Type soundness / noninterference / auditing?

• Plan to implement some variant of this language
  – Mainly to gain experience with how painful it is to use!
Open Questions

• This story seems just fine for *integrity*, but what about *confidentiality*?
  – Is there an "encryption" analog to "signatures" interpretation?

• Other practical issues:
  – Effects system?  
  – Channels and authentication… *More monads?*
  – Revocation/expiration of signed objects… *Nonces?*
  – Type inference?  
  – *Timestamps? Transactions?*
Related Work

• Authorization Logics:
  – Abadi, Burrows, Lampson, Plotkin "ABLP" [TOPLAS 1993]
    • somewhat ad hoc w.r.t. delegation and negation
  – Garg & Pfenning [CSFW 2006, ESORICS 2006]
    • a constructive modal logic that's very close to monomorphic DCC
  – Becker, Gordon, Fournet [CSFW 2007]

• Combining access control and information-flow:
  – Pistoia, Banerjee & Naumann [Oakland 2007, JFP 2005]
    • ACL induced information-flow policies, Stack-based access control
    • Jif-style dynamic principals and labels

• Connections to other modal logics?
  – Murphy et al. [LICS 2004]