Security-Typed Programming within Dependently-Typed Programming

Dan Licata
Joint work with Jamie Morgenstern

Carnegie Mellon University

Security-Typed Programming

- Access control: who gets access to what?
  - read a file
  - play a song
  - make an FFI call

- Information flow: what can they do with it?
  - post the file contents on a blog
  - copy the mp3
  - save the result in a database

Security-Typed Programming

- **Access control: who gets access to what?**
  read a file
  play a song
  make an FFI call

- Information flow: what do they do with it?
  post the file contents on a blog
  copy the mp3
  save the result in a database

Access Control

Alice

Read "/alice/secret.txt"

Desktop

Access control list (ACL) for /alice/secret.txt

Alice: rwad
Bob: rw
Admin: rlidwka

(slide by Kumar Avijit)

Access Control

Alice

Read "/alice/secret.txt"

Desktop

Access control list (ACL) for /alice/secret.txt

Alice: rwad
Bob: rw
Admin: rlidwka

Enforcement: Authentication + ACL lookup

(slide by Kumar Avijit)

Decentralized Access Control

Alice

Read “paper.pdf”

Digital library

- All students of members can read papers
- CMU is a member

CMU

- Alice is a student
- Charlie is a student
- …

(slide by Kumar Avijit)

Decentralized Access Control

Alice

Read “paper.pdf”

Digital library

- All students of members can read papers
- CMU is a member

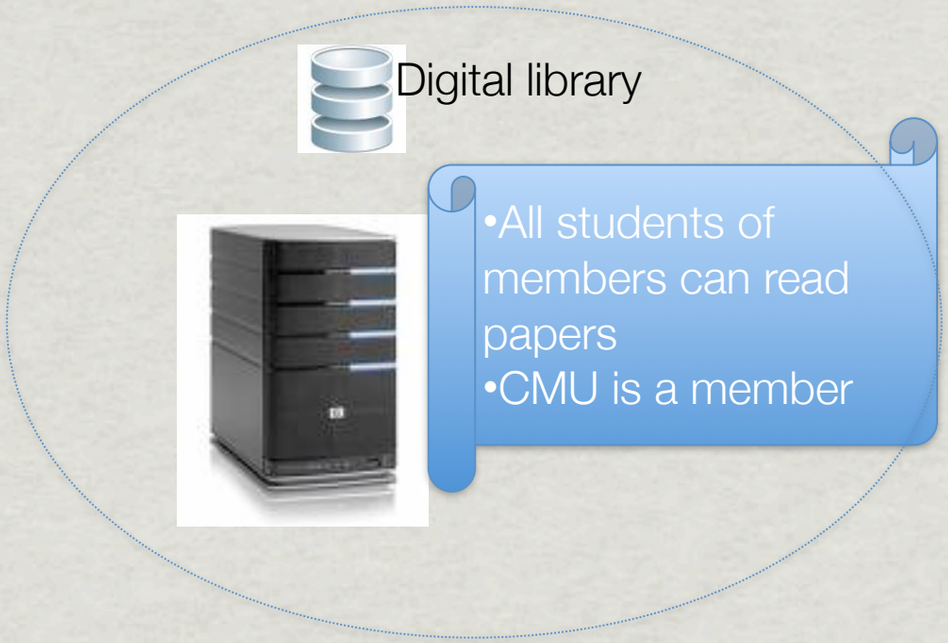
CMU

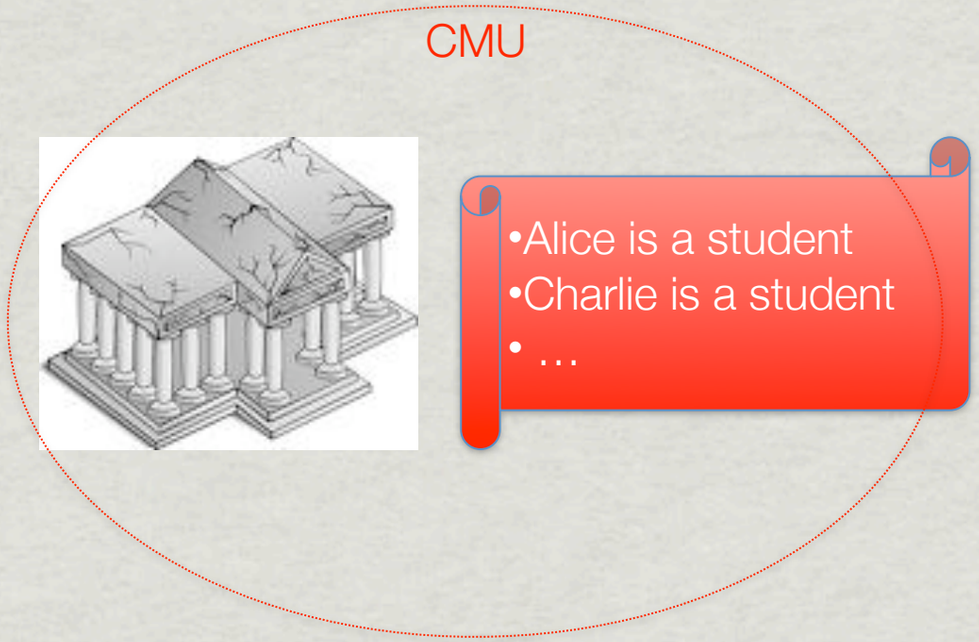
- Alice is a student
- Charlie is a student
- …

Need a mechanism to specify and enforce decentralized policies...

(slide by Kumar Avijit)

Authorization Logic





ACM **says** ∀ s:principal,
∀ i:principal,
∀ p:paper,
(member(i) ∧ i **says** student(s))
⊃ MayRead(s, p)

CMU **says** student(Alice)

Proof Carrying Authorization

[Appel+Felten]

Alice

Read "paper.pdf"

Digital library

- All students of members can read papers
- CMU is a member

CMU

- Alice is a student
- Charlie is a student
- …

(slide by Kumar Avijit)

Proof Carrying Authorization

[Appel+Felten]

Alice

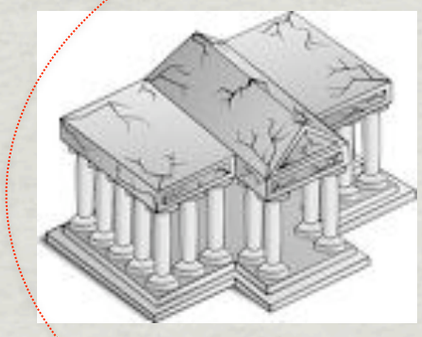
Read “paper.pdf”

p : mayread(Alice,paper.pdf)

Digital library

- All students of members can read papers
- CMU is a member

CMU



- Alice is a student
- Charlie is a student
- …

(slide by Kumar Avijit)

Proof Carrying Authorization

Alice

Read “paper.pdf”

p : mayread(Alice,paper.pdf)

Digital library

Evidence OK?

(Access granted)

(Runtime error)

(slide by Kumar Avijit)

Proof Carrying Authorization

Alice

Read “paper.pdf”

p : mayread(Alice,paper.pdf)

Digital library

Evidence OK?

(Access granted)

(Runtime error)

Can we ensure that runtime errors won’t happen?

(slide by Kumar Avijit)

An API for PCA

read : file → prin → proof → contents

e.g. read(paper.pdf,Alice,p)

An API for PCA

read : file → prin → proof → contents

e.g. read(paper.pdf,Alice,p)

- p might not be a well-formed proof

An API for PCA

read : file → prin → proof → contents

e.g. read(paper.pdf,Alice,p)

- p might not be a well-formed proof
- p might not be a proof of the right theorem!

Dependent Types!

```
read : (f : file) (k : prin)
       (p : proof(mayread(k,f))
    → contents
```

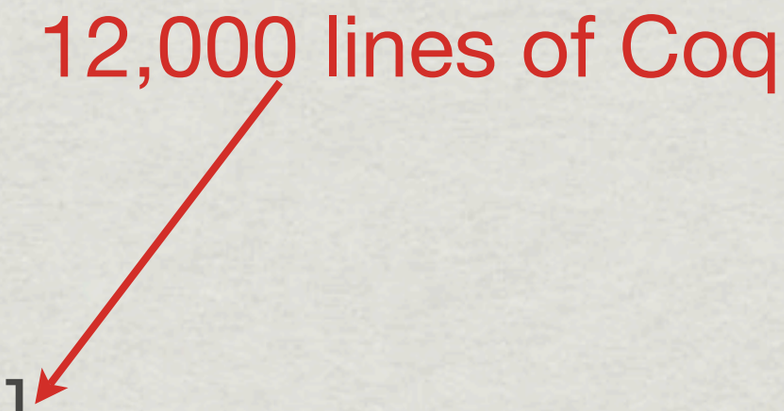
- p is well-formed by typing
- theorem is explicit in p's type

Dependent PCA

- PCML5 [Avijit+Harper]
- Aura [Jia,Vaughn,Zdancewik,et al.]
- Fine [Swamy et. al]

Dependent PCA

- PCML5 [Avijit+Harper]
- Aura [Jia,Vaughn,Zdancewik,et al.]
- Fine [Swamy et. al]



Dependent PCA

- PCML5 [Avijit+Harper]
- Aura [Jia,Vaughn,Zdancewik,et al.]
- Fine [Swamy et. al]

12,000 lines of Coq

20,000 lines of F#

Can we do

security-typed programming within an existing dependently-typed language

?

Security-typed Programming in Agda

1. Representing an authorization logic

2. Compile-time and run-time theorem proving

3. Stateful and dynamic policies

Security-typed Programming in Agda

1. **Representing an authorization logic**

2. Compile-time and run-time theorem proving

3. Stateful and dynamic policies

BL0 [Garg+Pfenning]

$$\frac{\Omega; \Delta; [] \xrightarrow{k} A}{\Omega; \Delta; \Gamma \xrightarrow{k0} k \text{ says } A} \text{ SAYSR}$$

BL0 [Garg+Pfenning]

$$\frac{\Omega; \Delta; [] \xrightarrow{k} A}{\Omega; \Delta; \Gamma \xrightarrow{k0} k \text{ says } A} \text{ SAYSR}$$

```
data _⊢R_ : (θ : Ctx) -> Propo+ (ictx θ) -> Set where
  saysR : ∀ {θ k A}
        -> (sayCtx θ k) ⊢ A
        -> θ ⊢R (k says A)
```

BL0 [Garg+Pfenning]

$$\frac{\Omega; \Delta; [] \xrightarrow{k} A}{\Omega; \Delta; \Gamma \xrightarrow{k0} k \text{ says } A} \text{ SAYSR}$$

```
data _⊢R_ : (θ : Ctx) -> Propo+ (ictx θ) -> Set where
  saysR : ∀ {θ k A}
        -> (sayCtx θ k) ⊢ A
        -> θ ⊢R (k says A)
```

BL0 [Garg+Pfenning]

$$\frac{\Omega; \Delta; [] \xrightarrow{k} A}{\Omega; \Delta; \Gamma \xrightarrow{k0} k \text{ says } A}\ \text{SAYSR}$$

```
data _⊢R_ : (θ : Ctx) -> Propo+ (ictx θ) -> Set where
  saysR : ∀ {θ k A}
        -> (sayCtx θ k) ⊢ A
        -> θ ⊢R (k says A)
```

Outline

1. Representing an authorization logic

2. Compile-time and run-time theorem proving

3. Stateful and dynamic policies

Theorem Prover

read(paper.pdf,Alice,p)

can be big and difficult to write out

Theorem Prover

read(paper.pdf,Alice,p)

can be big and difficult to write out

We implemented a theorem prover:

prove : (Θ : Ctx) (A : Prop) → Maybe (Θ ⊢ A)

Theorem Prover

read(paper.pdf,Alice,p)

can be big and difficult to write out

We implemented a theorem prover:

prove : **(n : nat)** (Θ : Ctx) (A : Prop) → Maybe (Θ ⊢ A)

search depth

Run-time Proving

```
tryRead : (Γ : Ctx) (p : prin)(f : file) → Maybe(string)
tryRead Γ p f = case (prove 15 Γ Mayread(f,p)) of
                  None => None
                  Some proof => Some (read p f proof)
```

Run-time Proving

```
tryRead : (Γ : Ctx) (p : prin)(f : file) → Maybe(string)
tryRead Γ p f = case (prove 15 Γ Mayread(f,p)) of
                    None => None
                    Some proof => Some (read p f proof)
```

prove is fancy version of “look up in ACL”

Run-time Proving

prove : (n:nat) (Θ : Ctx) (A : Prop) → Maybe (Θ ⊢ A)

```
tryRead : (Γ : Ctx) (p : prin)(f : file) → Maybe(string)
tryRead Γ p f = case (prove 15 Γ Mayread(f,p)) of
                    None => None
                    Some proof => Some (read p f proof)
```

prove is fancy version of “look up in ACL”

Compile-time Proving

Γpol a static (known at compile-time) policy:

Γpol = CMU says student(Alice) ::

ACM says ∀ s:principal,
∀ i:principal,
∀ p:paper,
(member(i) ∧ i says student(s))
⊃ MayRead(s, p) ::

...

Compile-time Proving

proof? : Maybe (Γpol ⊢ Mayread(Alice, paper.pdf))
proof? = prove 15 Γpol (Mayread(Alice, paper.pdf))

Compile-time Proving

proof? : Maybe (Γpol ⊢ Mayread(Alice, paper.pdf))

proof? = prove 15 Γpol (Mayread(Alice, paper.pdf))

Computes (defintional equality) to either None or Some(p)

Compile-time Proving

proof? : Maybe (Γpol ⊢ Mayread(Alice, paper.pdf))

proof? = prove 15 Γpol (Mayread(Alice, paper.pdf))

Computes (defintional equality) to either None or Some(p)

theProof : Γpol ⊢ Mayread(Alice, paper.pdf)

theProof = getSome proof?

Compile-time Proving

proof? : Maybe (Γpol ⊢ Mayread(Alice, paper.pdf))

proof? = prove 15 Γpol (Mayread(Alice, paper.pdf))

Computes (defintional equality) to either None or Some(p)

theProof : Γpol ⊢ Mayread(Alice, paper.pdf)

theProof = getSome proof?

Checks *at compile-time* that the theorem prover returned a proof

Compile-time Proving

```
isSome : {A : Set} -> Maybe A -> Bool
isSome (Some _) = True
isSome None = False

getSome : {A : Set} (s : Maybe A) -> {_ : Check (isSome s)} -> A
getSome (Some x) = x
getSome None {()}
```

theProof : Γpol ⊢ Mayread(Alice, paper.pdf)

theProof = getSome proof?

Checks *at compile-time* that the theorem prover returned a proof

Theorem Prover

```
proveRight : Nat -> (θ : Ctx) -> (A : Propo Pos (ictx θ)) -> Maybe (θ ⊢R A)
proveRight Z _ _ = None
proveRight (S n) θ (k0 says A) =
  proveNeutral n (sayCtx θ k0) A >>= λ y → Some (saysR y)
proveRight (S n) θ (∃i_{.(ictx θ)}{T} A) = or tryterms where
  terms = allTermsGen extraStrings (ictx θ) (▸ T)
  tryterms = ListM.map (λ t → map (∃R t) (proveRight n θ (substlast A t)))
                       terms
proveRight (S n) θ (⇣ A) = map (λ x → blurR x) (proveNeutral n θ A)
proveRight (S n) θ (A ∨ B) = (map ∨R1 (proveRight n θ A)) ||
                             (map ∨R2 (proveRight n θ B))
proveRight (S n) θ (A ∧ B) = proveRight n θ A >>= λ x →
                              map (λ y → ∧R x y) (proveRight n θ B)
proveRight (S n) θ ⊤ = Some ⊤R
```

Theorem Prover

```
proveLeft (S n) θ (A ⊃ B) C = (proveLeft n θ B C) >>= λ y →
    map (λ z → ⊃L z y) (proveRight n θ A)
proveLeft (S n) θ (a- A) (a- A') with atomEq A A'
proveLeft (S n) θ (a- A) (a- .A) | Some Refl = Some (init-)
... | None = None
proveLeft (S n) θ (a- _) _ = None
```

Theorem Prover

```
proveLeft (S n) θ (A ⊃ B) C = (proveLeft n θ B C) >>= λ y →
    map (λ z → ⊃L z y) (proveRight n θ A)
proveLeft (S n) θ (a- A) (a- A') with atomEq A A'
proveLeft (S n) θ (a- A) (a- .A) | Some Refl = Some (init-)
... | None = None
proveLeft (S n) θ (a- _) _ = None
```

Currently pretty slow: Agda's fault or ours?

Theorem Prover

```
stuck : (reader : Principal) →

        (MayRead reader paper.pdf :: [])
          ⊢ MayRead reader paper.pdf

stuck reader = getSome (prove 10)
```

Theorem Prover

```
stuck : (reader : Principal) →

        (MayRead reader paper.pdf :: [])
          ⊢ MayRead reader paper.pdf

stuck reader = getSome (prove 10)
```

gets stuck at termEq(reader,reader)

Theorem Prover

```
stuck : (reader : Principal) →

        (MayRead reader paper.pdf :: [])
          ⊢ MayRead reader paper.pdf

stuck reader = getSome (prove 10)
```

gets stuck at termEq(reader,reader)

Solution: reflection?

Outline

1. Representing an authorization logic

2. Compile-time and run-time theorem proving

3. Stateful and dynamic policies

Read with policy

read : (f : file) (k : prin)
(p : Γ ⊢ mayread(k,f))
→ string

Read with policy

$$\begin{aligned} \mathsf{read} : {} & (f : \mathsf{file})\ (k : \mathsf{prin}) \\ & (p : \Gamma \vdash \mathsf{mayread}(k,f)) \\ \rightarrow {} & \bigcirc\, \mathsf{string} \end{aligned}$$

Read with policy

```
read : (f : file) (k : prin)
       (p : Γ ⊢ mayread(k,f))
    → ◯ string
```

represents the policy; where does it come from?

Read with policy

read : (f : file) (k : prin)
(p : Γ ⊢ mayread(k,f))
→ ◯ string

represents the policy; where does it come from?

Want policies to be:

- dynamic: not known until run-time
- stateful: can change during execution (chown)

Indexed Monad

Represent computations with a type

[cf. HTT]

Indexed Monad

Represent computations with a type

[cf. HTT]

read : (f : file) (k : prin) (p : Γ ⊢ mayread(k,f))→ ◯ Γ string Γ

Indexed Monad

Represent computations with a type

[cf. HTT]

read : (f : file) (k : prin) (p : Γ ⊢ mayread(k,f)) → ◯ Γ string Γ

chown : (f : file) (k1 k2 : prin)
(p : (Γ,owns(k1,f)) ⊢ maychown(k1,f))
→ ◯ (Γ,owns(k1,f)) string (Γ,owns(k2,f))

Indexed Monad

$$\begin{aligned} \mathsf{read} : {} & (f : \mathsf{file})\ (k : \mathsf{prin}) \\ & (p : \Gamma \vdash \mathsf{mayread}(k, f)) \\ \rightarrow {} & \bigcirc\ \Gamma\ \mathsf{string}\ \Gamma \end{aligned}$$

Indexed Monad

supposed to be running as k

read : (f : file) (k : prin)
(p : Γ ⊢ mayread(k,f))
→ ◯ Γ string Γ

Indexed Monad

running as k

```
read : (f : file) (k : prin)
       (p : Γ ⊢ mayread(k,f) & as(k))
  →  ◯ Γ string Γ
```

Indexed Monad

running as k

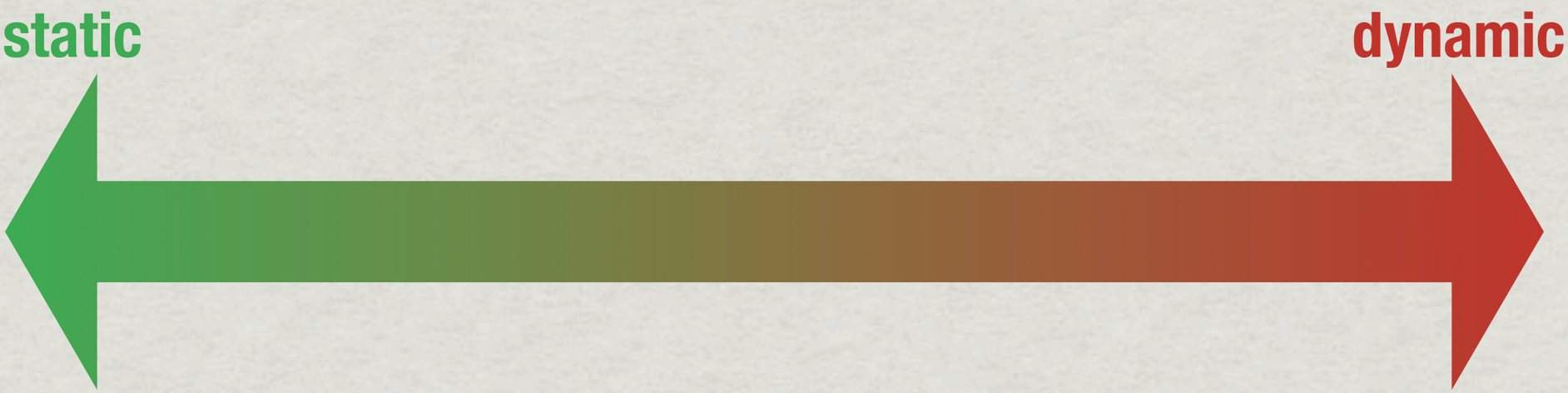
```
read : (f : file) (k : prin)
       (p : Γ ⊢ mayread(k,f) & as(k))
  →  ◯ Γ string Γ
```

```
sudo : (f : file) (k1 k2 : prin)
     →  Γ,as(k1) ⊢ maysu(k1,k2)
     →  ◯ (Γ,as(k2)) C (Γ',as(k2))
     →  ◯ (Γ,as(k1)) C (Γ',as(k1))
```

More examples [ICFP'10, to appear]

- file access control (more details)
- located computation
- combination with information flow
- conference management server with several phases (submission, reviewing, notification, …)

Sliding scale



- Guess the policy
- Prove consequences statically
- Failures only at edges

- Do all proving at run-time
- Type system ensures you make the right run-time checks and handle failures

Summary

Can do security-typed programming within a DTPL

- Indexed inductive definition to represent proofs
- Theorem prover to discharge proof obligations, run at compile-time (getSome) and run-time
- Indexed monad to manage stateful+dynamic policies

Feature Requests

How could a DTPL better support this application?

- Speed (theorem prover)
- Reflection (prover works well at extremes  but not in the middle)
- Binding+scope (logic)

Thanks for listening!

paper + code at http://www.cs.cmu.edu/~drl