

A Logical Foundation for Authorizing Code Execution

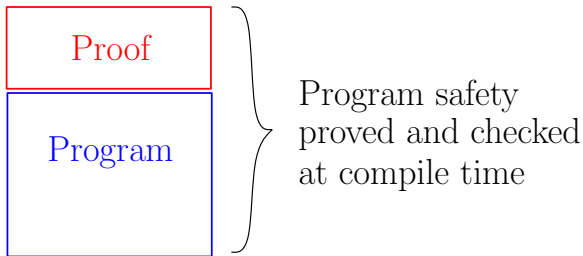
Maja Frydrychowicz

supervised by Prof. Brigitte Pientka

30.08.2007

- Authorizing code execution: must manage access to resources
- Authorization logic is an effective tool for designing access control policies
- **Problem:** How to model the **interaction** between programs and their governing access control policy?
- **Our approach:** a programming language whose type system directly incorporates an authorization logic

- **Well-typed** program guaranteed to respect rules of given access control policy
- Policy enforcement is part of program development



Example: Confused Deputy Problem

Trusted code (K1) performs restricted operation on behalf of untrusted code (K2).

```
(let box
  u = (deleteFile)K1
  in
  u[proof] "someFile"
end)K2
```

Principals: $K1, K2$

Atomic Permissions: WRITE

Restricted Function: deleteFile requires WRITE

Policy: $(K1 \text{ says WRITE}) \supset \text{WRITE}$

- Can we formally verify that program respects given policy?
- At what points does program interact with policy?
- Fixed policy impractical: dynamic policy updates
- Take direct advantage of desirable properties guaranteed by authorization logic

- Modular type system consists of 2 interacting levels:

COMPUTATION + AUTHORIZATION

- Use contextual modalities to reason about truth relative to propositions in authorization logic.
- Track program's effects on policy with code signing and principal statements

Typing Judgments & Syntax

Delegation Chain $\delta ::= \cdot \mid \delta/K$

Authorization Context $\Psi ::= \cdot \mid \Psi, p : A$

Computation Context $\Gamma ::= \cdot \mid \Gamma, x : T \mid \Gamma, u :: T[\Psi]$

2 Typing Judgments $\delta; \Psi; \Gamma \vdash e : T \quad \Psi \vdash s : A$

Authorization-level expressions:

$s, s_1, s_2 ::= p \mid \lambda p. s \mid s_1 s_2 \mid s_K \mid \text{bind } p = s_1 \text{ in } s_2 \text{ end}$

Computation-level expressions:

$e, e_1, e_2 ::= x \mid u[\sigma] \mid \text{fn } x. e_K \mid e_1 e_2 \mid \text{rec } x. e_K \mid e_K$
 $\mid \text{box } \Psi. e \mid \text{let box } u = e_1 \text{ in } e_2 \text{ end}$

$$\frac{\delta/K; \Psi; \Gamma \vdash e : T}{\delta; \Psi; \Gamma \vdash e_K : T} \text{ sign}$$

$$\frac{\Psi, (\delta \text{ says } \Psi') \vdash \sigma : \Psi'}{\delta; \Psi; \Gamma, u : T[\Psi'] \vdash u[\sigma] : T} \text{ ctxhyp}$$

Confused Deputy Example Revisited

Restricted Function: `deleteFile : (T1 → T2)[WRITE]`

Policy Ψ : `(K1 says WRITE) \supset WRITE`

```
(let box
  u = (deleteFile)K1
in
  u[proof] "someFile"
end)K2
```

The expression will not type-check because:

`(K1 says WRITE) \supset WRITE`, `K2 says (K1 says WRITE) $\not\supset$ WRITE`

- Contextual Modal Type Theory (Nanevski, Pfenning, Pientka 2007)
- Non-Interference in Constructive Authorization Logic (Garg, Pfenning 2006)
- A Modal Foundation for Secure Information Flow (Miyamoto, Igarashi 2004)
- Proof-Carrying Code (Necula 1997)
- A Type Discipline for Authorization Policies (Fournet, Gordon, Maffeis 2005)
- Stack Inspection: Theory and Variants (Fournet, Gordon 2003)

Results So Far:

- Local soundness and completeness
- Type preservation

To Do:

- Dynamic policy updates
- Extending non-interference
- Erasure-based translation to an ordinary pure functional language
- (...)

Summary of Contributions

- Modular type system relates computation to a formally specified access control policy
- Type system may be extended with a more expressive authorization logic
- Method for tracking program's interaction with policy via principal statements
- Program adherence to security policy established at compile time.