Encoding Information Flow in Haskell

Steve Zdancewic

Peng Li

University of Pennsylvania

To appear: CSFW 2006

Secure Information Flow

- Problem: protecting confidential information manipulated by computer systems
  - Governments want this to protect confidential documents (e.g. "NSA phone records acquisitions")
  - Businesses want to protect trade secrets yet still collaborate with business partners
  - Health regulations may require careful handling of medical records
  - Individuals may want to protect their private information (pin #s, financial records, e-mail, etc.) stored on their home computers.
- Security of data and infrastructure is critical

[Trust in Cyberspace, Schneider et al. '99]

Current State of the Art

- Access control
  - File permissions / Capabilities / Stack Inspection / …
- Encryption
  - Authentication / Secure channels / …

- Neither of these provides *end-to-end* guarantees:
  - It's hard to control *propagation* of information after access has been granted
  - It's hard to *compute* with encrypted objects

- Complementary mechanism: *Language-based Security*
  - Provide language features to implement secure software

Information-flow Policies

- There must be some way to distinguish "secret" from "public" information: *labels*

- Use a lattice: (Higher = "more confidential")

  [Denning 70's]

  - Order: $l_1 \leq l_2$
  - Meet: $l_1 \wedge l_2$
  - Join: $l_1 \vee l_2$

- Example: LOW ≤ MEDIUM ≤ HIGH

Language-based Security

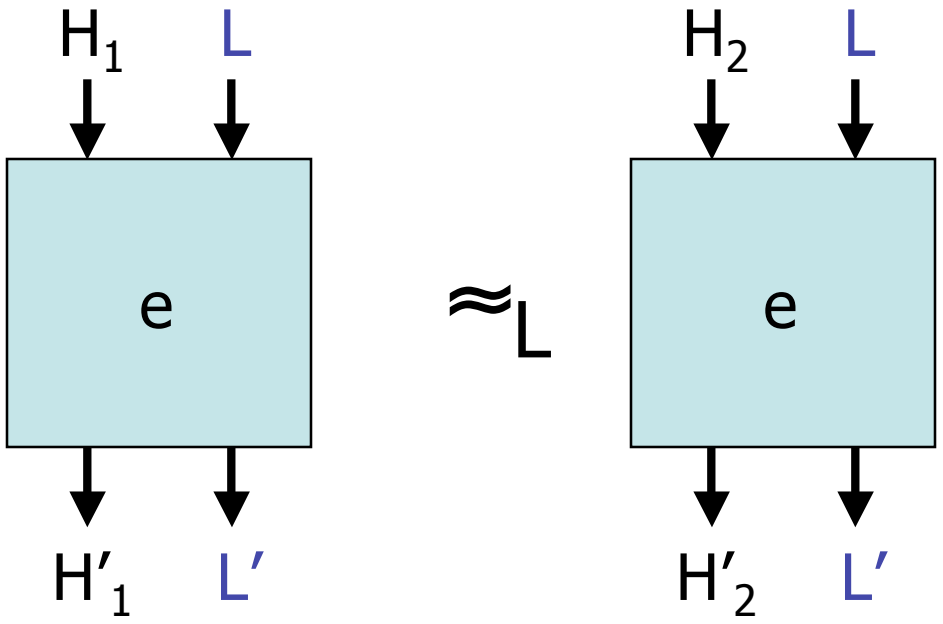
- Use the labels as (parts of) the types in the language.
  [VSI'96,…]
- Track the labels in the type system to ensure that the policy is enforced & rule out *implicit flows*:

$$\frac{\Gamma \vdash e : l_1 \quad \Gamma \vdash e_1 : l_2 \quad \Gamma \vdash e_2 : l_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : l_1 \vee l_2}$$

- Key observation: static analysis of control flow graph is essential
  - dynamic enforcement is too coarse

Noninterference

[Rey'78,GM'82,…]



- Related to parametricity and representation independence
- Proved by:
  - Logical relations
  - Bisimulation techniques
  - Soundness of nonstandard operational semantics

Existing Security-typed Languages

- Jif [Myers et al.]
  - Variant of Java with "decentralized label model"
  - Handles many features: classes, exceptions, state, dynamic policies, etc.
  - Software: Battleship games [Myers et al.], JifPoker [Sabelfeld], JPMail [Hicks et al.] , Jif Web Servlet Framework [Chong et al]

- FlowCaml [Simonet and Pottier]
  - Dialect of Ocaml
  - Strong support for polymorphism and label inference

Barriers to Adoption

- Special-purpose languages (like Jif and FlowCaml):
  - Require wholesale adoption of a new language (they're supposed to enforce end-to-end policies)
  - Hard to safely interface with existing code
  - Re-train programmers (costly)
- Overspecialization?
  - Hard-wired label model (may not fit all applications)
  - Hard to add new features like:
    - Dynamic policies, different interpretations of the security labels
    - Fancy new downgrading or declassification mechanisms [Sands, …]
- Too big of a hammer? (Or even the wrong tool?)
  - Security is only one desirable feature among many (functionality, performance, usability, etc.)
  - Hopefully only a small fraction of a large system will be relevant to security
  - Twisting the entire development process to accommodate one design criterion may be too much

Encoding Information Flow

- Main idea: use a general purpose language (Haskell) rather than a domain specific language.
  - Encode the information-flow policies using the constructs provided by the general purpose language.
  - Embed this sublanguage in Haskell

- Benefits:
  - It's easier to interface with existing (Haskell) code.
  - Haskell programmers don't have to learn a new language.
  - Making security a "library" or "design pattern" promotes modularity.
  - Developing Haskell libraries is easier than writing compilers
  - Eliminating a special purpose compiler means that the security features can be tailored to the application's needs.
  - More readily experiment with security features

Additional Considerations

- Challenges:
  - Encoding the information-flow policies (labels, etc.)
  - Soundly enforcing the policy, dealing with implicit and explicit information flows
- Questions:
  - What language features do we need?
  - How useful is it in practice?
  - What security guarantees can we obtain?

Encoding Information Flow in Haskell

- Embedding a secure sublanguage in Haskell
  - Encoding the label lattice
  - Monads
  - Hughe's "Arrows" (generalize monads) [Hughes '98]
  - A definition of `FlowArrow`
  - Adding new features (declassification)
- Example implementation & code demo
- Caveats & future directions

Encoding the Label Lattice

- *Dynamically* (encode the lattice as Haskell terms)
  - Simple: just use a datatype
  - Flexible: easy to program new lattices, accommodate "advanced" features like dynamic policies
  - How do you tie the labels to a "static" analysis of the embedded language?
- *Statically* (encode the lattice as Haskell types)
  - Previous work shows how to encode label lattice using polymorphic types [Tse & Zdancewic]
  - Perhaps give stronger guarantees
  - Less flexible

- Here, we take the dynamic approach.

A Haskell Typeclass for Labels

```
class (Eq a) => Lattice a where
   label_top      :: a
   label_bottom   :: a
   label_join     :: a → a → a
   label_meet     :: a → a → a
   label_leq      :: a → a → Bool
```

- Plus, stipulations that these operations meet the expected axioms
- Example: Definition of TriLabel

A Security Sublanguage of Haskell

- Basic idea: Provide an abstract type `Protected a`
  - Encapsulates a "secure computation that creates a value of type `a`"
  - Abstraction is important: It shouldn't be possible to forge these values

- Provide combinators to build more complex programs from simple `Protected` building blocks.

- Natural question: can we construct `Protected a` as a monad?

Monads in Haskell

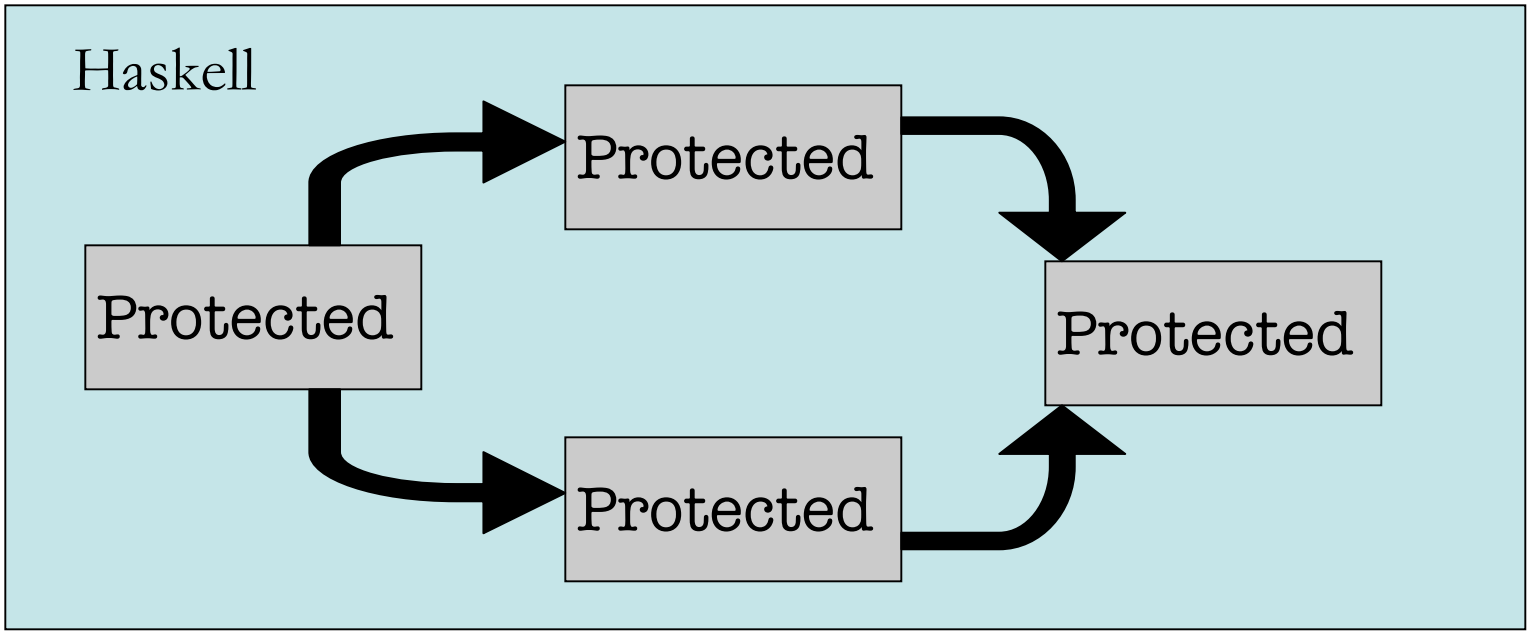
```
class Monad m where
    return      :: a → m a
    (>>=)       :: m a → (a → m b) → m b
```

- Highlights:
  - `return` "lifts" a pure expression into the monad
  - bind `>>=` provides sequential composition
  - Must satisfy several monad laws (e.g. composition is associative)

- We can think of a monad as a simple programming language with at least "`skip`", "`;`", and some monad-specific primitive operations.

Intuition

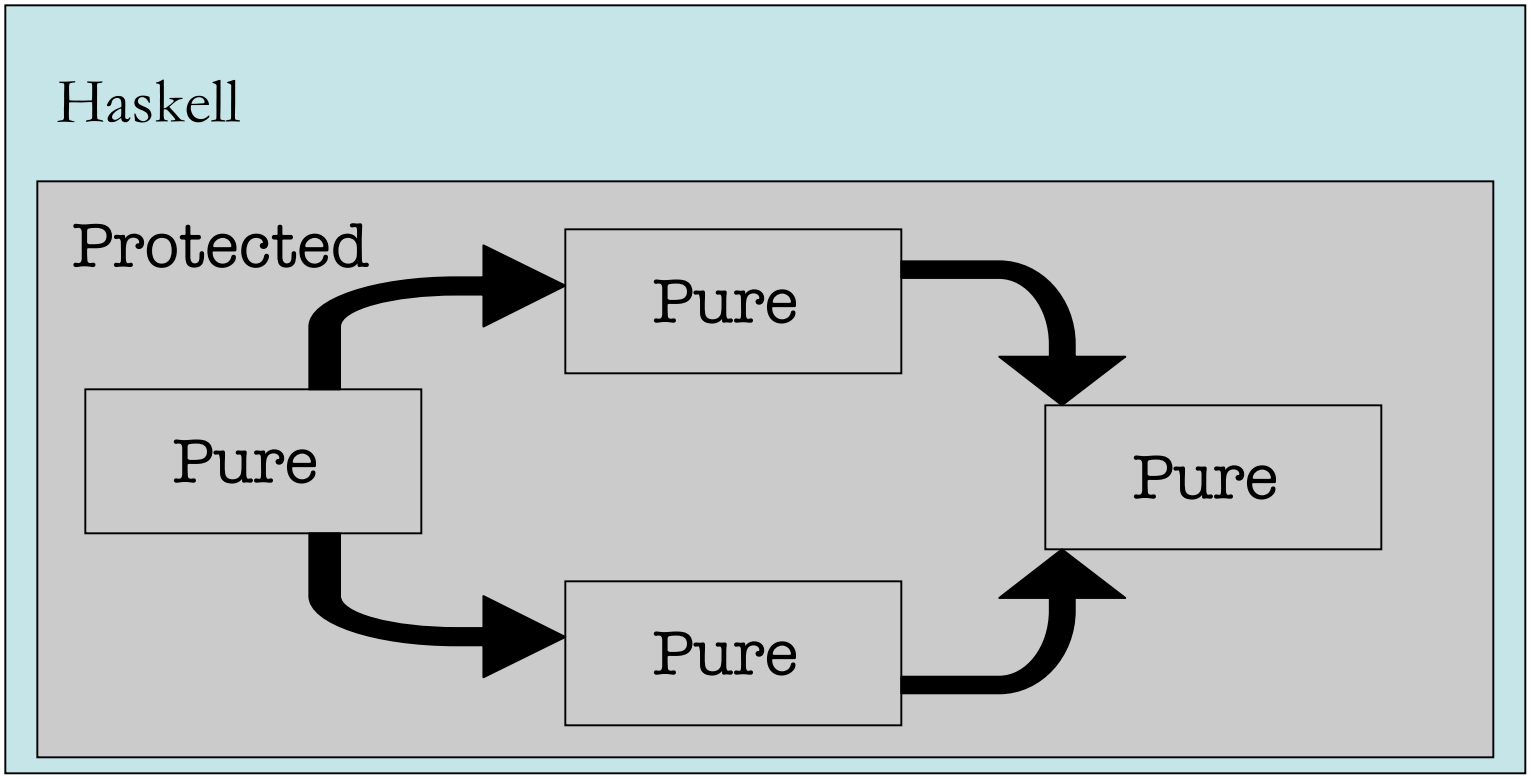
- Monads let us build an embedded sublanguage in which "basic blocks" are built from `return` and `(>>=)`
- The rest of the control-flow graph (i.e. the wiring between the basic blocks) is created using Haskell-level constructs.



- Problem: Can't statically analyze the control-flow graph.

Solution

- Use a more general abstraction that lets us represent the control-flow graph explicitly



Arrows

[Hughes '98]

```
-- Sequential Composition
class Arrow a where
   pure      :: (b → c) → a b c
   (>>>)     :: a b c → a c d → a b d
   first     :: a b c → a (b,d) (c,d)
   (***)     :: a b c →  a d e  →  a (b,d) (c,e)

-- Conditionals
class ArrowChoice a where
   left      :: a b c → a (Either b d)(Either c d)
   (|||)     :: a b d → a c d →  a (Either b c) d
```

Haskell's Arrow Syntax

- GHC provides convenient syntactic sugar for arrow programming:
  - The "do" notation:

```
liftA2 :: Arrow a => (b → c → d)
                  → a e b → a e c → a e d
liftA2 op f g = proc x → do
    y <- f -< x
    z <- g -< x
    returnA -< y `op` z
```

Summary So Far

- We can easily encode a lattice using Haskell's datatypes.
- The Arrow abstraction provides a framework for defining embedded sublanguages in Haskell.
  - The CFG of the embedded language can be "statically" analyzed by the Haskell level

- The idea: assign an information-flow type system to arrows

Information-flow Types for Arrows

$$\frac{}{\emptyset \;\vdash\; \texttt{pure}\ f \;:\; l \to l}$$

$$\frac{\Phi_1 \;\vdash\; c_1 : l_1 \to l_2 \qquad \Phi_2 \;\vdash\; c_2 : l_3 \to l_4}{\Phi_1 \cup \Phi_2 \cup \{l2 \leq l3\} \;\vdash\; c_1 \;\texttt{>>>}\; c_2 : l_1 \to l_4}$$

Arrow Typing Rules (cont'd)

$$\frac{\Phi_1 \vdash c_1 : l_1 \rightarrow l_2 \qquad \Phi_2 \vdash c_2 : l_3 \rightarrow l_4}{\Phi_1 \cup \Phi_2 \vdash c_1 \;|||\; c_2 : (l_1 \wedge l_3) \rightarrow (l_2 \vee l_4)}$$

- These typing rules are easy to implement in Haskell.
  - In practice: make `FlowArrow` an Arrow transformer
- `Protected a` is an instance of `FlowArrow`
- Haskell desugars "`if - then - else`" into an appropriate combination of Arrow combinators:
  - Expected typing rule for "`if`" is obtained for free!

Tagging Data

- We need a way to explicitly label the data in the embedded language

$$\frac{}{\emptyset \vdash \mathsf{tag}\ l : l \to l}$$

- When composed on the right, tag constrains the output of an arrow.
- When composed on the left, tag constrains the input to an arrow

- Demo code: FlowArrow.hs Test.hs

Adding Declassification

- Add a new arrow: `declassify`

$$\{l_1 \leq l_{user}\} \quad |\text{- } \texttt{declassify}\ l_1\ l_2 : l_1 \rightarrow l_2$$

- l_{user} is a label constant that represents the privilege level of the user

- Provide an abstract type `Priv` that encapsulates the declassification capabilities of the ambient Haskell code.
  - For example: `Priv = Pr lattice`

Program Certification

- Putting it all together:
  - Given a control flow graph c constructed from `FlowArrow` combinators
  - Given a concrete lattice $\mathcal{L}$
  - Given the privilege level, `Pr L_user`

- Certify that all of the label constraints on the program are met:

$$\frac{\Phi \vdash c : l_1 \rightarrow l_2 \qquad \mathcal{L} \models \Phi[\texttt{L\_user}/l_{user}] \qquad \texttt{label\_bottom} \leq l_1 \qquad l_2 \leq \texttt{label\_bottom}}{\mathcal{L} \vdash \texttt{cert (Pr L\_user)}\ c}$$

Demo Code

- Simple "bidding" server
  - Guests can place bids, but can't see the highest bid
  - Administrators can see (and reset) highest bid
- Authentication database
  - Maps username and password to `Priv` object
  - Is itself a protected object with label HIGH
- Uses declassification:
  - Assumes that the I/O channel is LOW observable

Trusted Computing Base & Security

- What do we need to trust?
  - Implementation of `FlowArrow`
  - Implementation of `cert`
  - Parts of the program that can manufacture `Priv` objects

- Security guarantee (conjecture):
  - Let e_1 and e_2 have type `Protected a`
  - Let H[-] be a well-typed Haskell context with no way to obtain a `Priv` object and yielding a `Bool`
  - Then: $H[e_1] \rightarrow^* v$ iff $H[e_2] \rightarrow^* v$

- Question: How do we refine this to account for "privileged" code?

Caveats and Disclaimers

- Debugging may be more difficult:
  - Checking occurs only for code that is executed.
- Haskell provides `Unsafe.performIO`
- For more stateful programming, one needs to adapt the type system
  - Haskell is mostly pure anyway
- Arrow laws: our `FlowArrow` is conservative
- Our simple dynamic privilege model may not be sufficient
  - Untrusted code could duplicate or replay privileges
  - This is an instance of the standard problems with capability-based security mechanisms
  - Use "lifetimes" or "one-shot" privileges… or other revocation strategies

Conclusions & Future Work

- Embedding security-typed languages in Haskell
- Easily able to accommodate basic information-flow language properties
- Conjectures:
  - It's easy to experiment with different label models, declassification features, etc.
- Future directions:
  - Proof of soundness?
  - Can we make a more precise FlowArrow type system?
  - Seeing how far Haskell's type system can be pushed to encode information-flow properties
  - Can we build practical examples solely in Haskell?

Thanks!

What have we gained?

- Haven't we just replaced barrier to adopting a security-oriented language with adopting Haskell?
  - Yes, but… there are already more Haskell programmers than Jif programmers
  - Haskell's features are intended to be more general purpose, so they're more likely to be adopted by mainstream languages