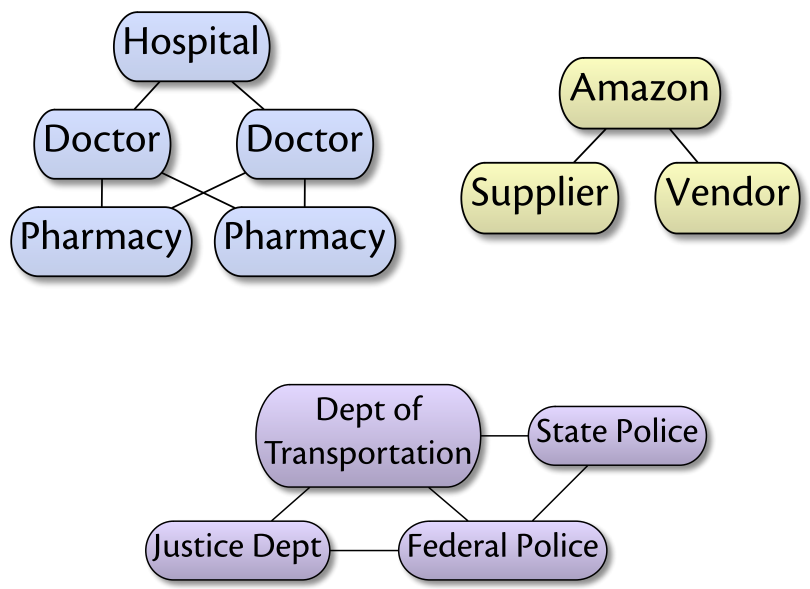
Constructive Security Using Information Flow Control

Andrew Myers

Cornell University

What is computer security?

- *Past*: can an attacker control my computer?
- *Future*: do networked systems sharing information provide security and privacy despite limited trust?
  - web applications, mashups
  - social networking platforms
  - medical information systems
  - government information systems
  - supply chain management
  - the Internet



Security requirements

Amazon.com Privacy Notice:

...We reveal only the last four digits of your credit card numbers when confirming an order. Of course, we transmit the entire credit card number to the appropriate credit card company during order processing.

...third-party Web sites and advertisers, or...advertising companies...sometimes use technology to send...advertisements that appear on our Web site directly to your browser. They automatically receive your IP address...

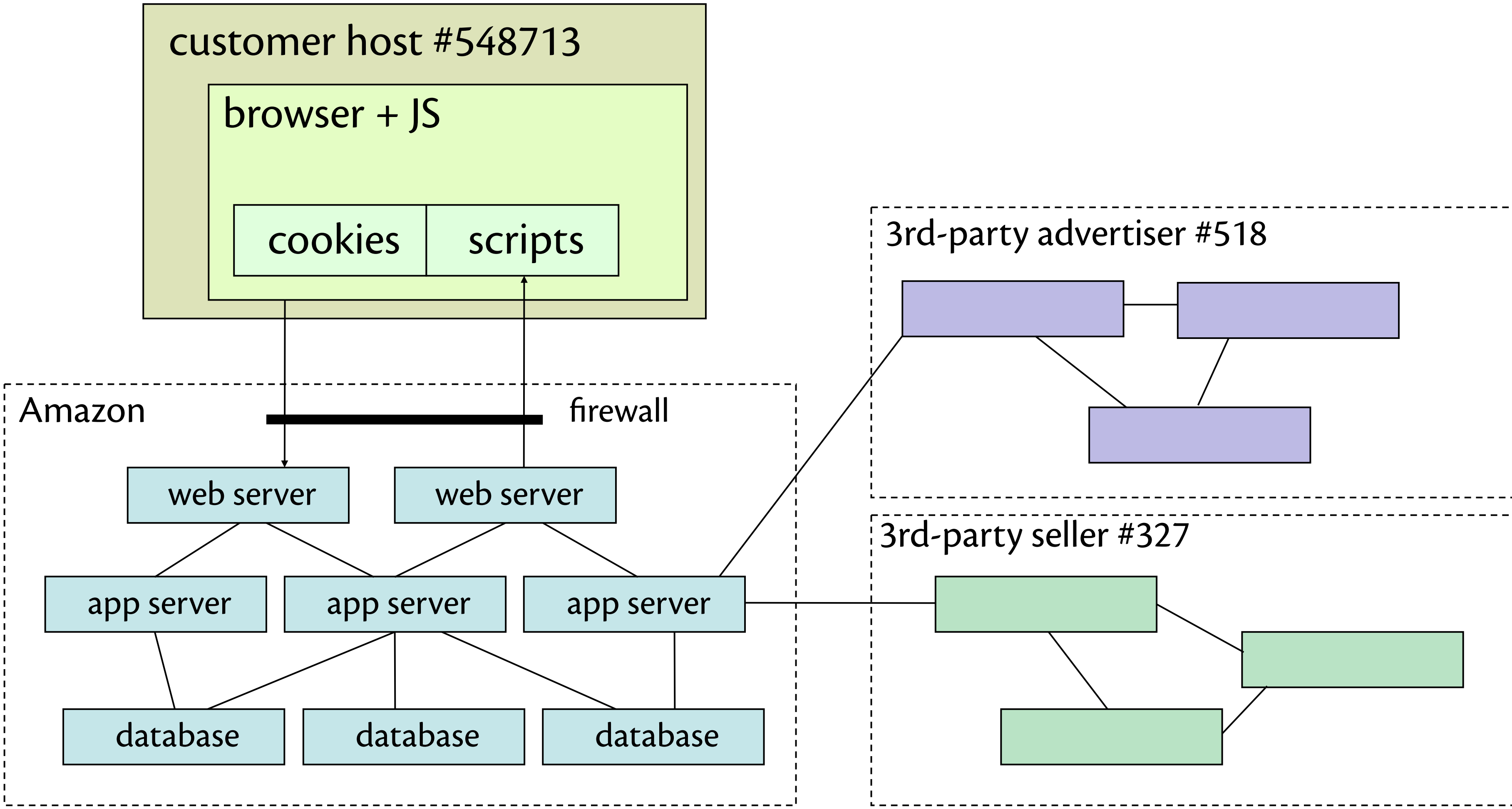
...Examples of the information we collect and analyze include...[IP]address used to connect your computer to the Internet; login; e-mail address; password; computer and connection information...plug-in types and versions, operating system, and platform; purchase history...the full [URL]clickstream...the phone number you used to call our 800 number...cookies...we may use...JavaScript to measure and collect session information, including...scrolling, clicks, and mouse-overs...

...Sometimes we send offers to selected groups of Amazon.com customers on behalf of other businesses. When we do this, we do not give that business your name and address. If you do not want to receive such offers, please adjust your Customer Communication Preferences.

...We release account and other personal information when...appropriate to comply with the law; enforce or apply our Conditions of Use and other agreements; or protect the rights, property, or safety of Amazon.com, our users, or others.

...Lots of promises about confidentiality and integrity...

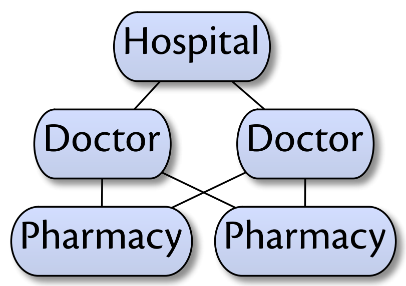
Requirements → mechanisms?

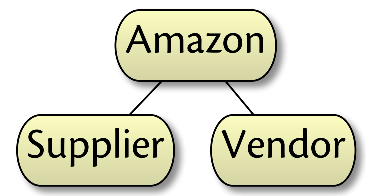


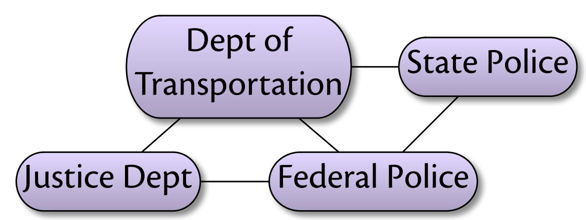
How does Amazon know this evolving system containing many nodes, code from many sources meets their legal obligations?

Cooperation with distrust

- *Past*: can an attacker control my computer?
- *Future*: do networked systems sharing information provide security and privacy despite limited trust?
  - web applications, mashups
  - social networking platforms
  - medical information systems
  - government agencies
  - supply chain management
  - the Internet

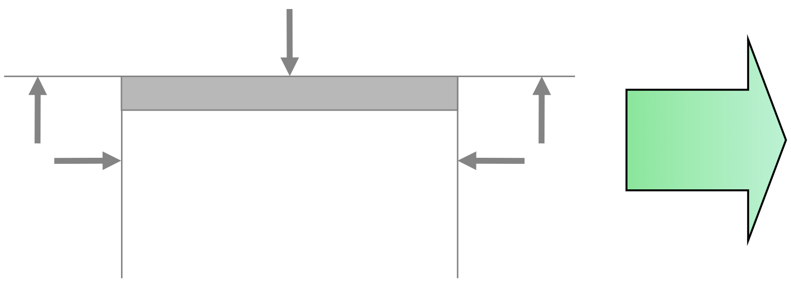


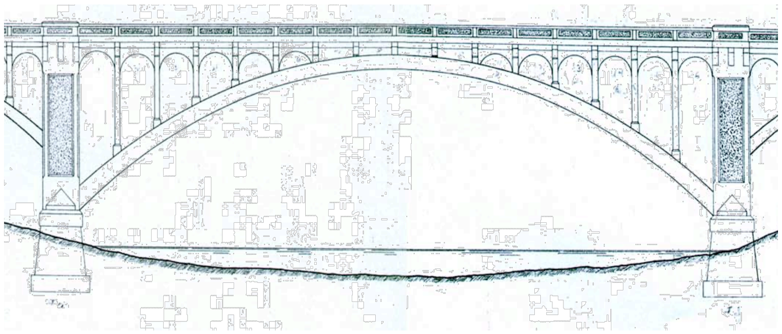




Security: bridges vs. software

- Bridges fail rarely (post-arch)
  - Assurance derived from construction process

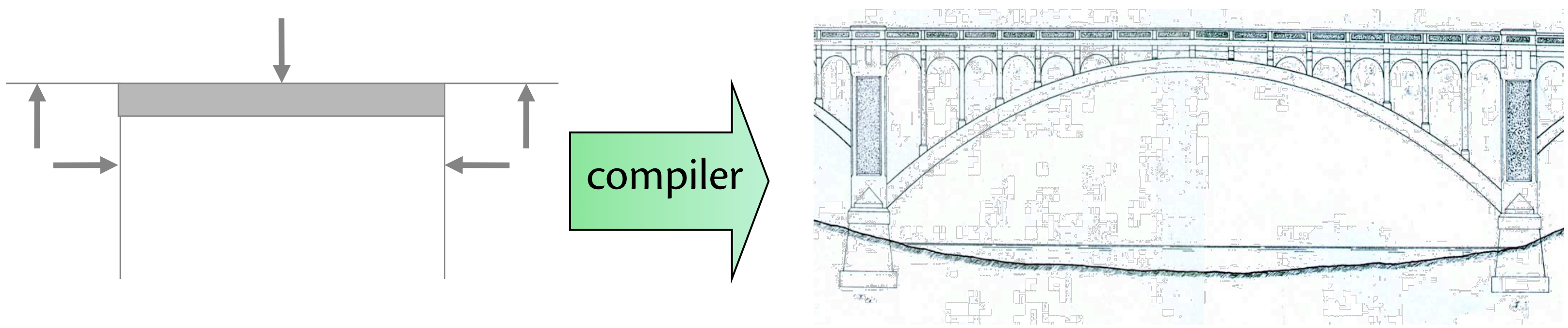




- Software violates security/privacy (frequently)
  - Assurance is weak at best
  - Much “destructive” security research

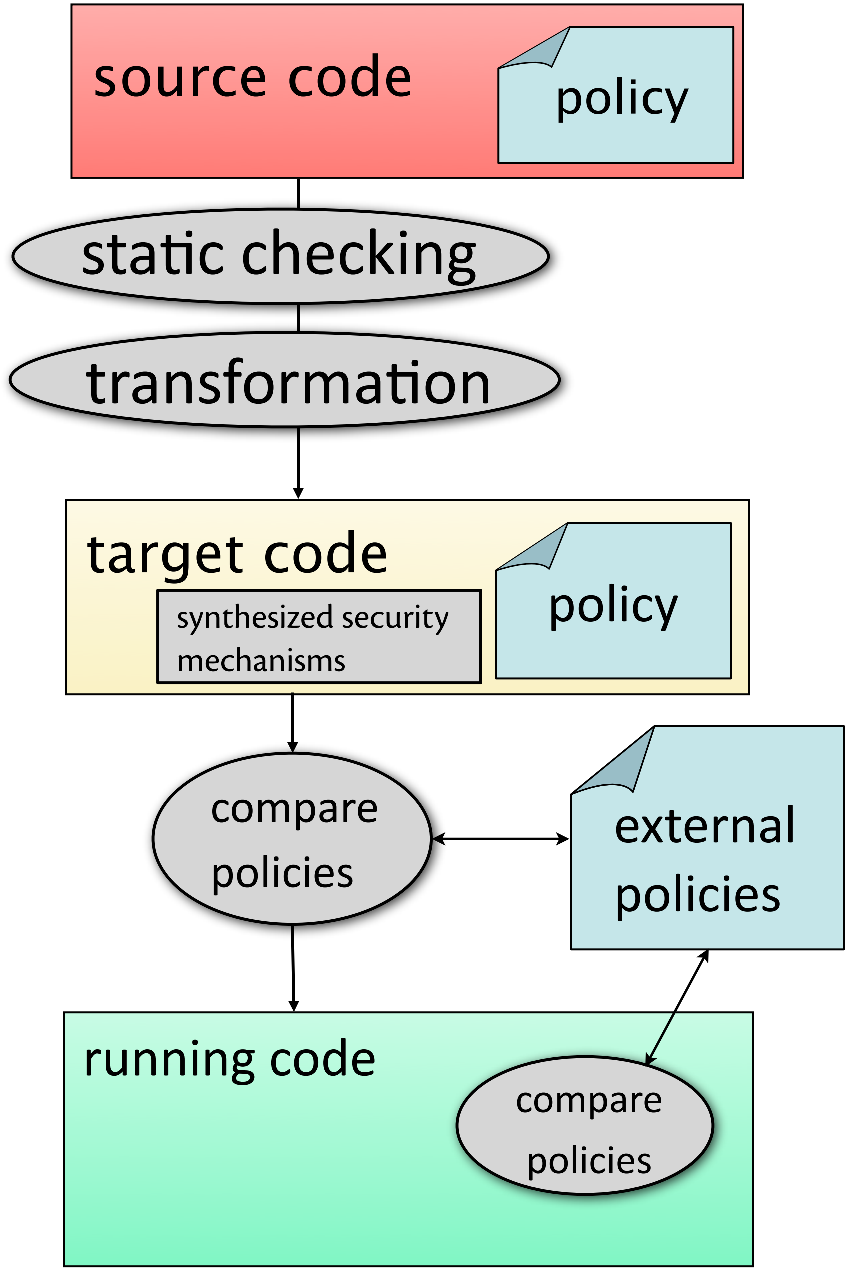
Constructive security?

- *Idea*: build secure systems with:
  - explicit, declarative security policies capturing security requirements
  - higher-level language-based abstractions
- Compiler, runtime *automatically* employ mechanisms to achieve security and performance
  - synthesizing implementation-level mechanisms (access control, partitioning, replication, encryption, signatures, logging, ...)
- Security by construction!



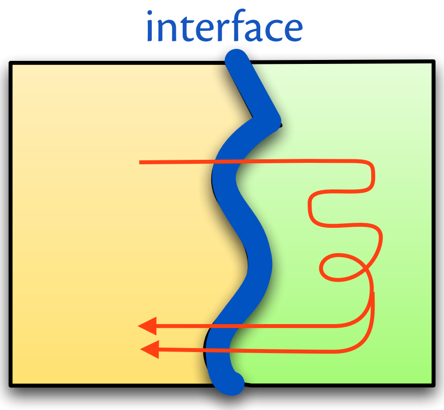
Language-based security

- Developer writes code in a safe language (e.g., Jif) with explicit security policies
- Software construction process checks policies are enforced, adds run-time enforcement mechanisms
- Can verify target code to ensure policy enforcement
- Policies exposed for checking against rest of system at load time and run time



Policies and end-to-end security

- System-wide, end-to-end enforcement of policies for information security ⇒ need *compositional* policies



code modules,
network nodes,
services,
…

- **Information flow** policies on interfaces constrain end-to-end behavior

  ⇒ are compositional

  ⇒ enable raising the level of abstraction

Plan

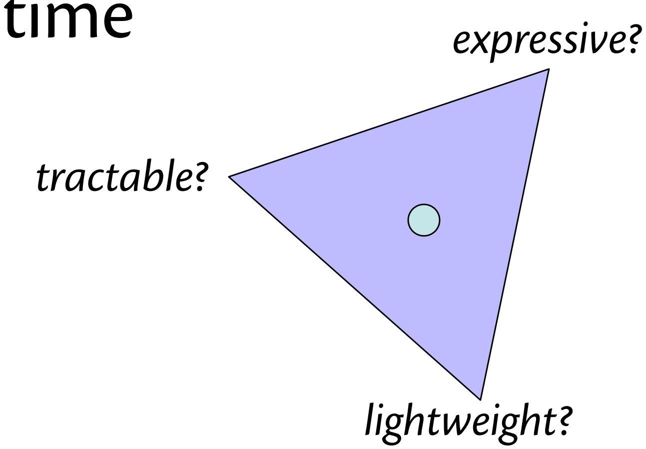
1. **Jif:** **J**ava + **i**nformation **f**low control

2. **Swift:** synthesizing secure web applications

3. **Fabric:** a distributed platform for secure computation, sharing, and storage

Jif: A security-typed language

- Jif = **J**ava + **i**nformation **f**low control [POPL99]
  - Types include explicit (but simple) security policies
  - Enforcement: compile-time and run-time

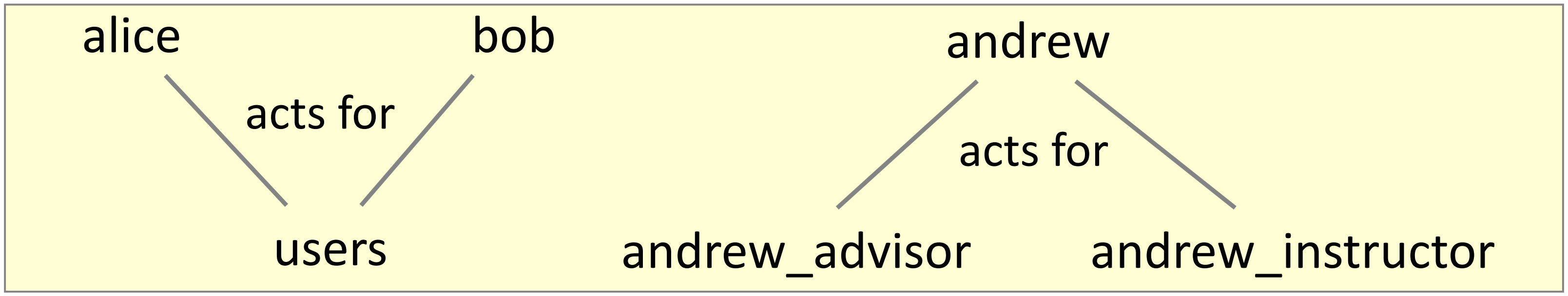


- Trust and access control:
  principals and **authority**
- Information flow: **decentralized labels**

Principals in Jif

A **principal** is an abstraction of authority and trust

- represents users, groups, roles; privileges; access rights; host nodes and other system components.
- **acts-for relation** p ≽ q means p can do whatever q can. "q trusts p". (related to speaks-for in authentication logic [e.g., ABLP93])



- Top, bottom principals:
  "acts for everyone" = ⊤ ≽ p ≽ ⊥ = "acts for no one"
- Principals form a lattice with meet (∧) and join (∨).

Programming with authority

- Code can run with the **authority** of a principal.

```
class C authority(Alice) {
    int m() where authority(Alice) {
        f(); // use authority of Alice
    }
    int f() where caller(Alice) { … }
}
```

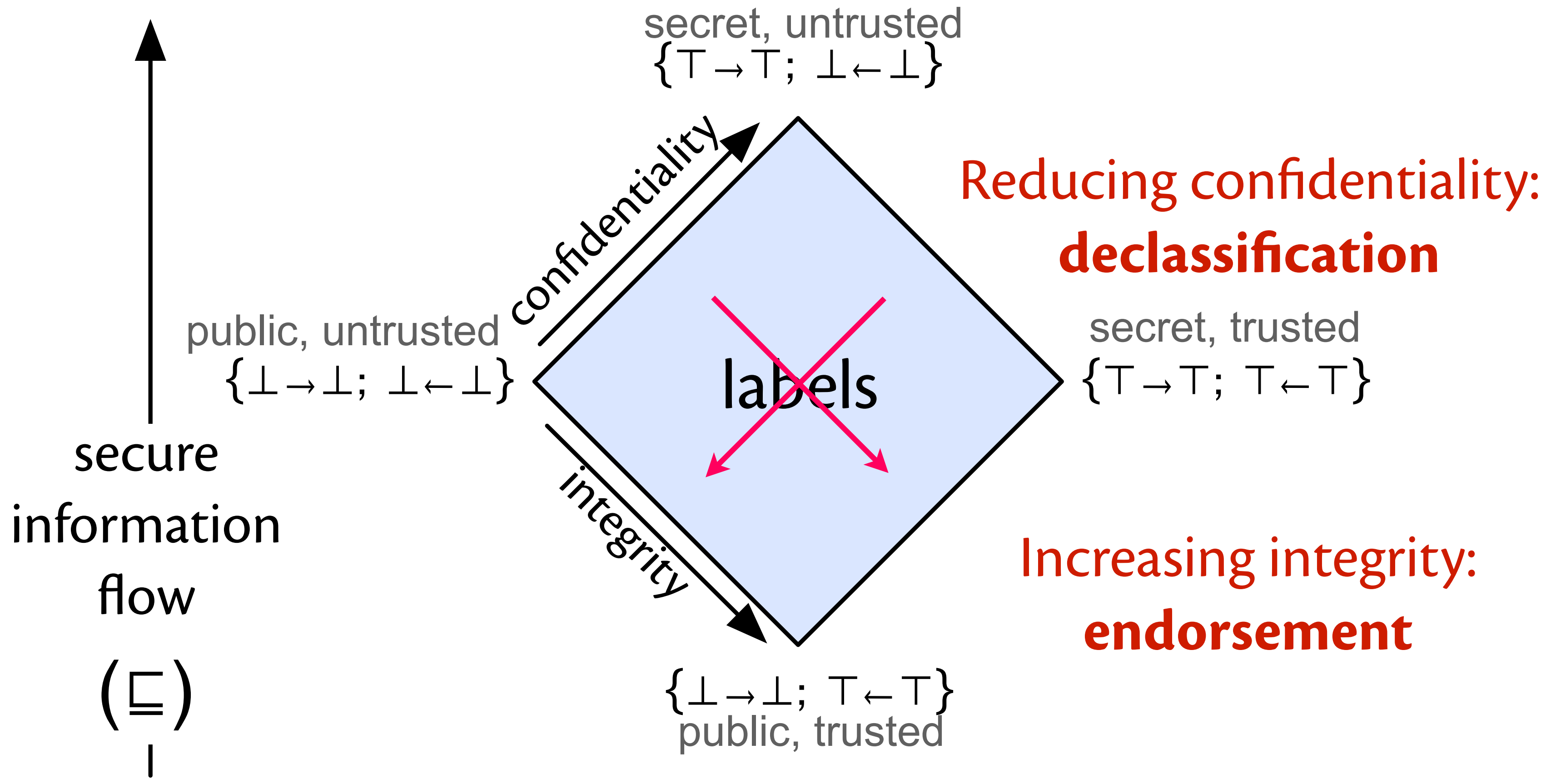
- Can be used to implement access control

Decentralized labels

- Confidentiality policies: **u→p**
  - u is the **owner** of the policy (a principal), p is a **reader**
  - meaning: u trusts p to learn information and not leak it
  - e.g., **Bob → Alice** means Bob trusts Alice (and Bob) to learn information about the data
- Integrity policies: **u←p**
  - meaning: u trusts p not to influence the information in a way that damages it
  - p is a **writer** of the information
- Decentralized label: set of owned policies

e.g., {Alice→Bob; Alice ←Alice}

Decentralized label space



- Application-specific downgrading is needed by real applications
- Dangerous, so controlled in Jif by requiring authority (trusted code only) and integrity (for **robust declassification**)

Information security policies as types

- Confidentiality labels: `int{Alice→Bob} a;`

  "Alice says only Bob (&Alice) can learn a"

- Integrity labels: `int{Alice←Alice} a;`

  "Alice says only Alice can affect a"

Combined: `int{Alice→Bob ; Alice←} a;`

- End-to-end static checking of flow $L_1 \rightarrow L_2$: $L_1 \sqsubseteq L_2$?

```
int{Alice→} a1, a2;
int{Bob←} b;
int{Bob←Alice} c;
```

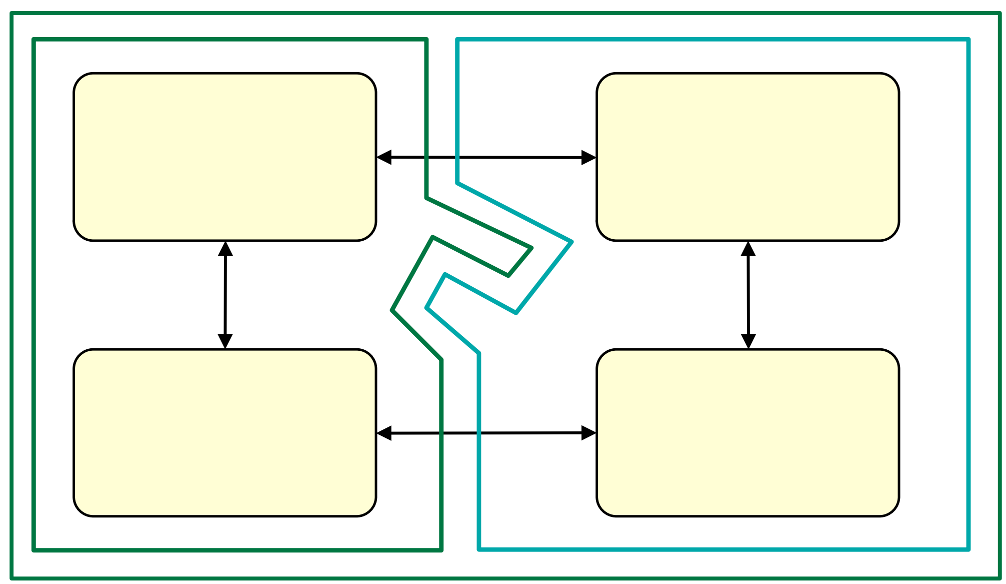
"Bob says only Alice (& Bob) can affect c"

Insecure	Secure
`b = a1;`	`a1 = a2;`
`b = c;`	`a1 = b;`
	`a1 = c;`
	`c = b;`

But: ok if Alice ≽ Bob

Information flow control as type checking

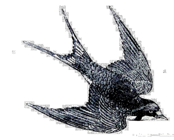
- Jif label checking is type checking in a nonstandard type system: compositional!
- End-to-end security: noninterference (termination-insensitive)
  - caveat: proved for simplified models
  - challenges: objects, dynamic labels and principals, dependent types, parameterized types, exceptions, ...



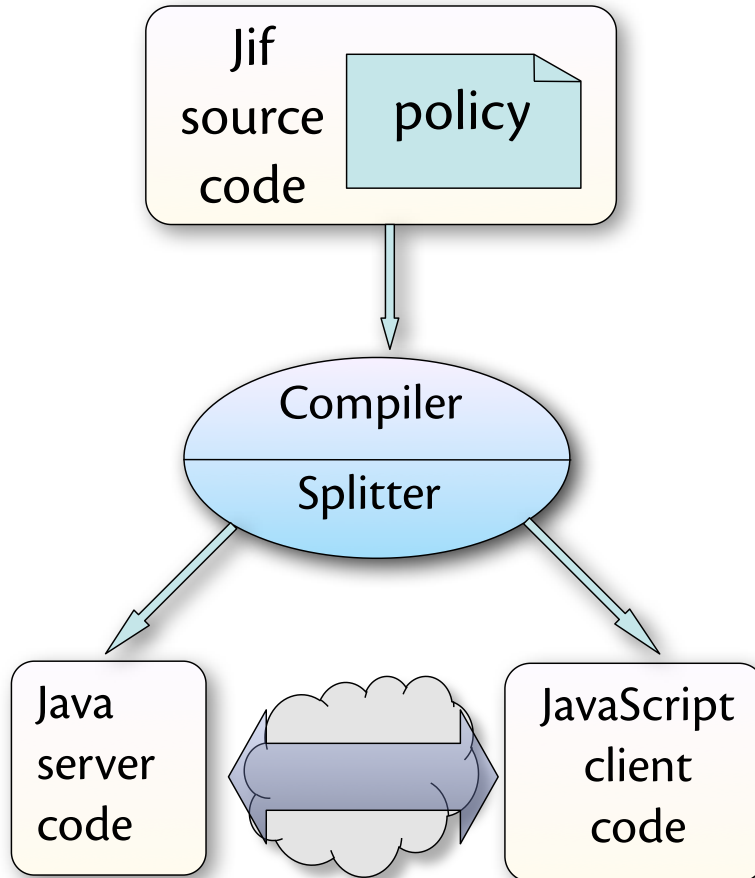
Secure web applications?

- Ubiquitous, important, yet insecure
  - Cross-site scripting, SQL injection, information leakage, etc.
- Development methods make security assurance hard
  - Distributed system in multiple languages
    - Client: CSS, XHTML, JavaScript, Flash
    - Server: PHP, ASP, Ruby, SQL
  - Ajax/Web 2.0: Complex JavaScript UIs generating HTTP requests

*Symantec Internet Security Threat Report 2007

Swift 

- A programming system that makes secure, interactive web applications easier to write [SOSP 07]
- A higher-level programming model: **one** program in **one** language automatically split by the compiler
- Security by construction:
  - automatically partitioning code and data based on decentralized labels
- Automatic performance optimization



Guess-the-number

Random number between 1 and 10

Secret Number: 7

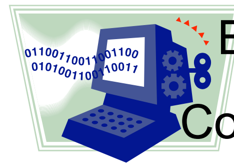
Tries: 3

Take a Guess!

ENTER

(You have 3 chances)

Guess-the-number

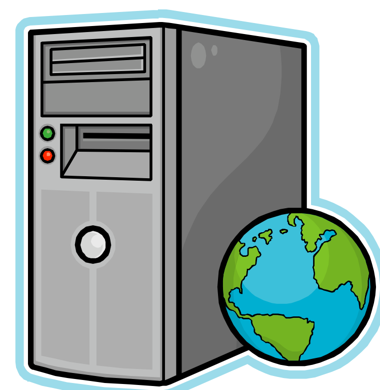


Bounds Check

Compare Guess

Secret Number: 7

Tries: 3



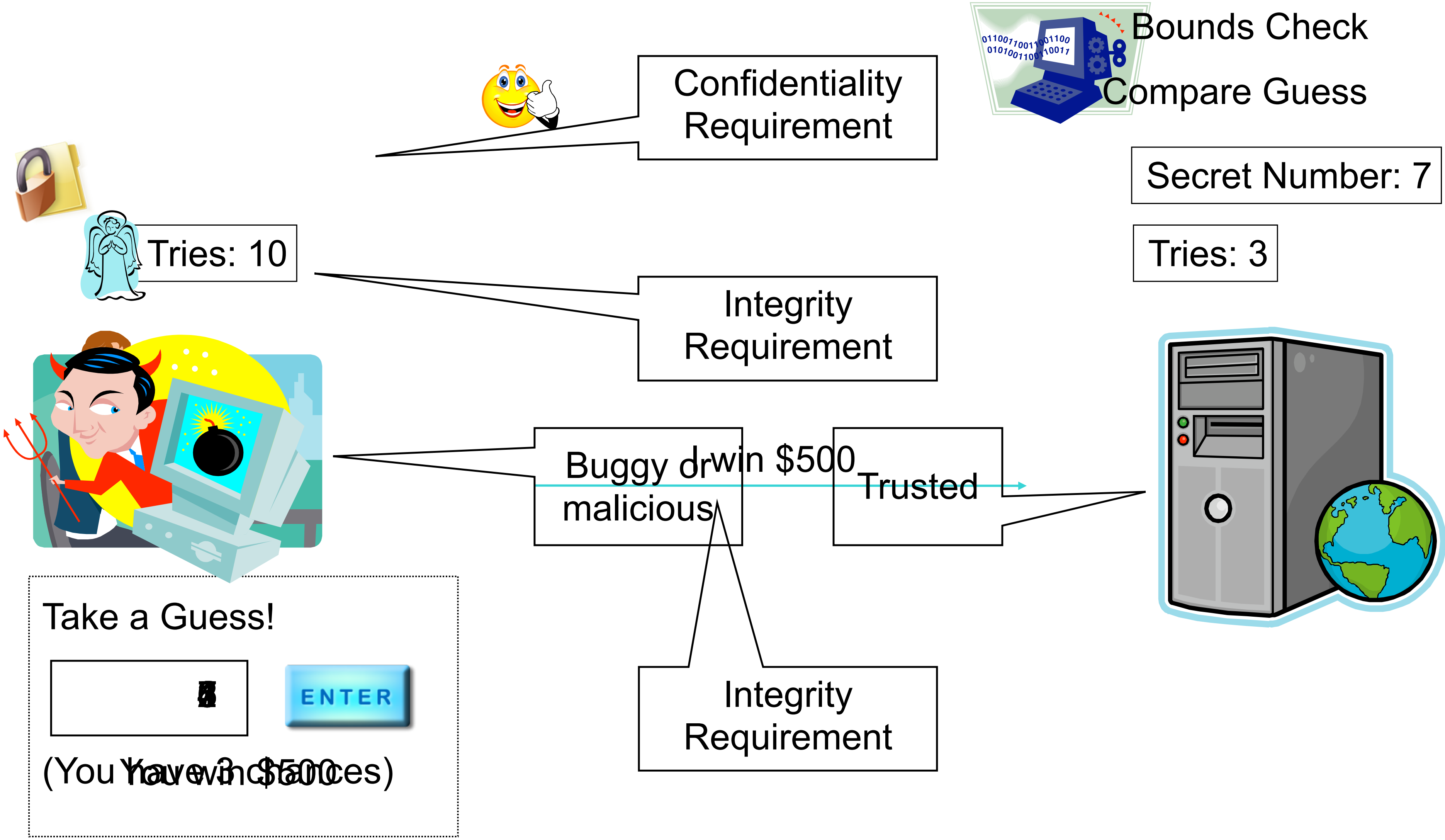
Client		Server
	6 →	
	← Try Again	
	12 →	
	← Out of range	
	4 →	
	← Try Again	
	7 →	
	← You win $500	

Take a Guess!

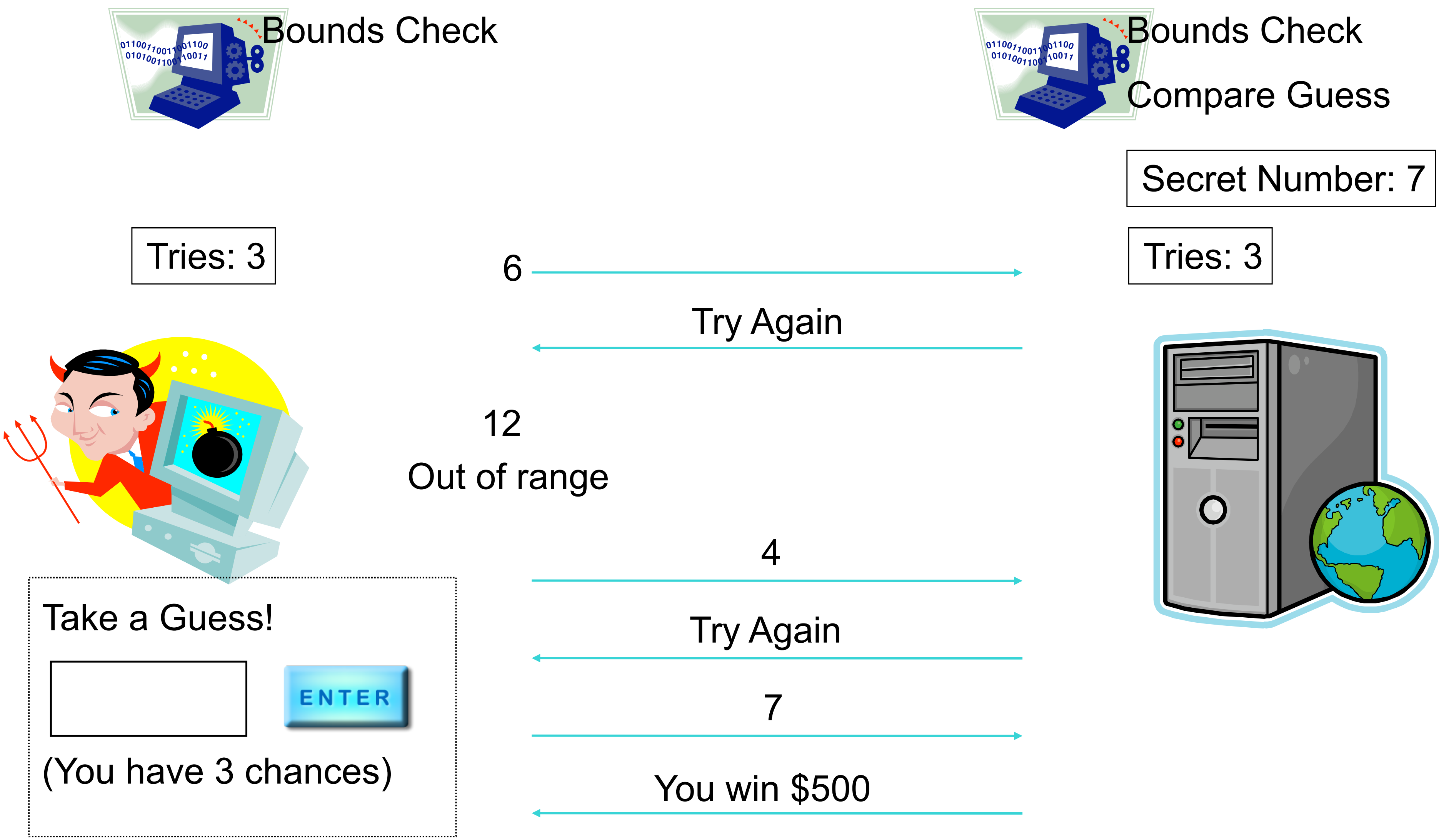
(You have 3 chances)

You win $500

Guess-the-number



A secure optimal split



Swift Guess-the-number

```
int secret;
int tries;
...
void makeGuess (int guess))
{

  if (guess >= 1 && guess <= 10) {










  } else {
    message.setText("Out of range:" + guess);
  }
}
```

called from a Listener

input validation

check fails

Swift Guess-the-number

```
int secret;
int tries;
...
void makeGuess (int guess)
{

  if (guess >= 1 && guess <= 10) {
    boolean correct = (guess == secret);

     if (tries > 0 && correct) {
         finishApp("You win $500!");
     }









  } else {
    message.setText("Out of range:" + guess);
  }
}
```

compare with stored secret

successful guess

Swift Guess-the-number

```
int secret;
int tries;
...
void makeGuess (int guess)
{

  if (guess >= 1 && guess <= 10) {
    boolean correct = (guess == secret);

    if (tries > 0 && correct) {
        finishApp("You win $500!");
    } else {
        tries--;
        if (tries > 0)
            message.setText("Try again");
        else
            finishApp("Game over");
    }
  } else {
    message.setText("Out of range:" + guess);
  }
}
```

compare with stored secret

unsuccessful guess

Automatic partitioning

```
int secret;
int tries;
...
void makeGuess (int guess)
{

  if (guess >= 1 && guess <= 10) {
    boolean correct = (guess == secret);

    if (tries > 0 && correct) {
        finishApp("You win $500!");
    } else {
        tries--;
        if (tries > 0)
           message.setText("Try again");
        else
           finishApp("Game over");
    }
  } else {
    message.setText("Out of range:" + guess);
  }
}
```

```
int secret;
int tries;
...
void makeGuess (int guess)
{

  if (guess >= 1 && guess <= 10) {
    boolean correct = (guess == secret);

    if (tries > 0 && correct) {
      finishApp("You win $500!");
    } else {
      tries--;
      if (tries > 0)
        message.setText("Try again");
      else
        finishApp("Game over");
    }
  } else {
    message.setText("Out of range: " + guess);
  }
}
```

Security policies

- Swift adds two built-in principals: **server, client**
- Application can define more principals (Alice, Bob, …)

Alice → Bob	= Alice permits Bob to learn info
Alice ← Bob	= Alice permits Bob to affect info

```
int{ server→server ; server←server} secret;

int{ server→client ;  server←server} tries;

int{server→client} display;

display = secret;
```

Rejected at compile time

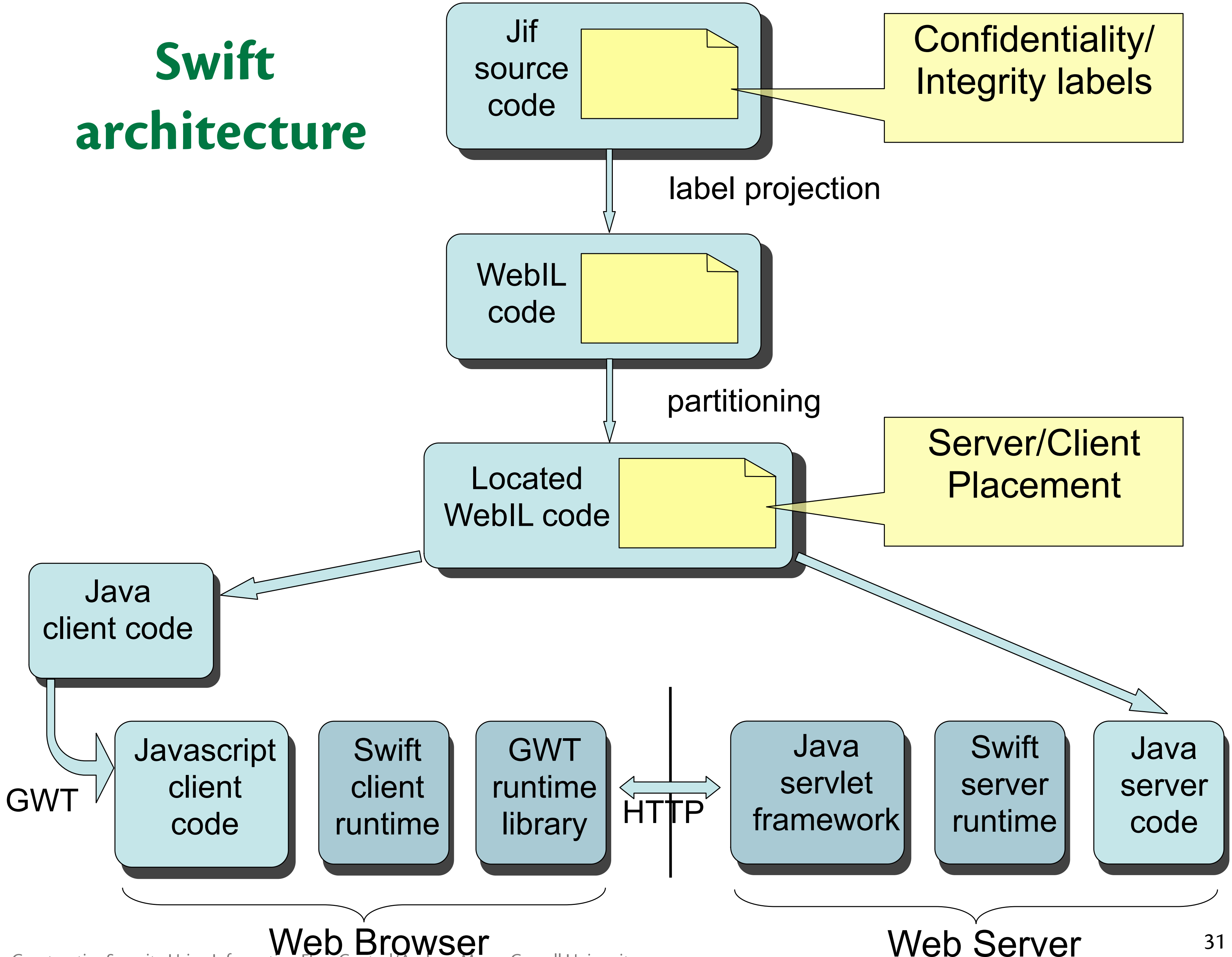
```
int{server→server; server←server} secret;
int{server→client; server←server} tries;
...

{
 endorse (guess, {server←client} to {server←server})
  if (guess >= 1 && guess <= 10) {
    boolean correct = declassify(guess == secret,
                        {server→server} to {server→client});
    if (tries > 0 && correct) {
        finishApp("You win $500!");
    } else {
        tries--;
        if (tries > 0)
           message.setText("Try again");
        else
           finishApp("Game over");
    }
  } else {
    message.setText("Out of range:" + guess);
  }
}
```

client guess within bounds can be treated as trusted: checked endorse

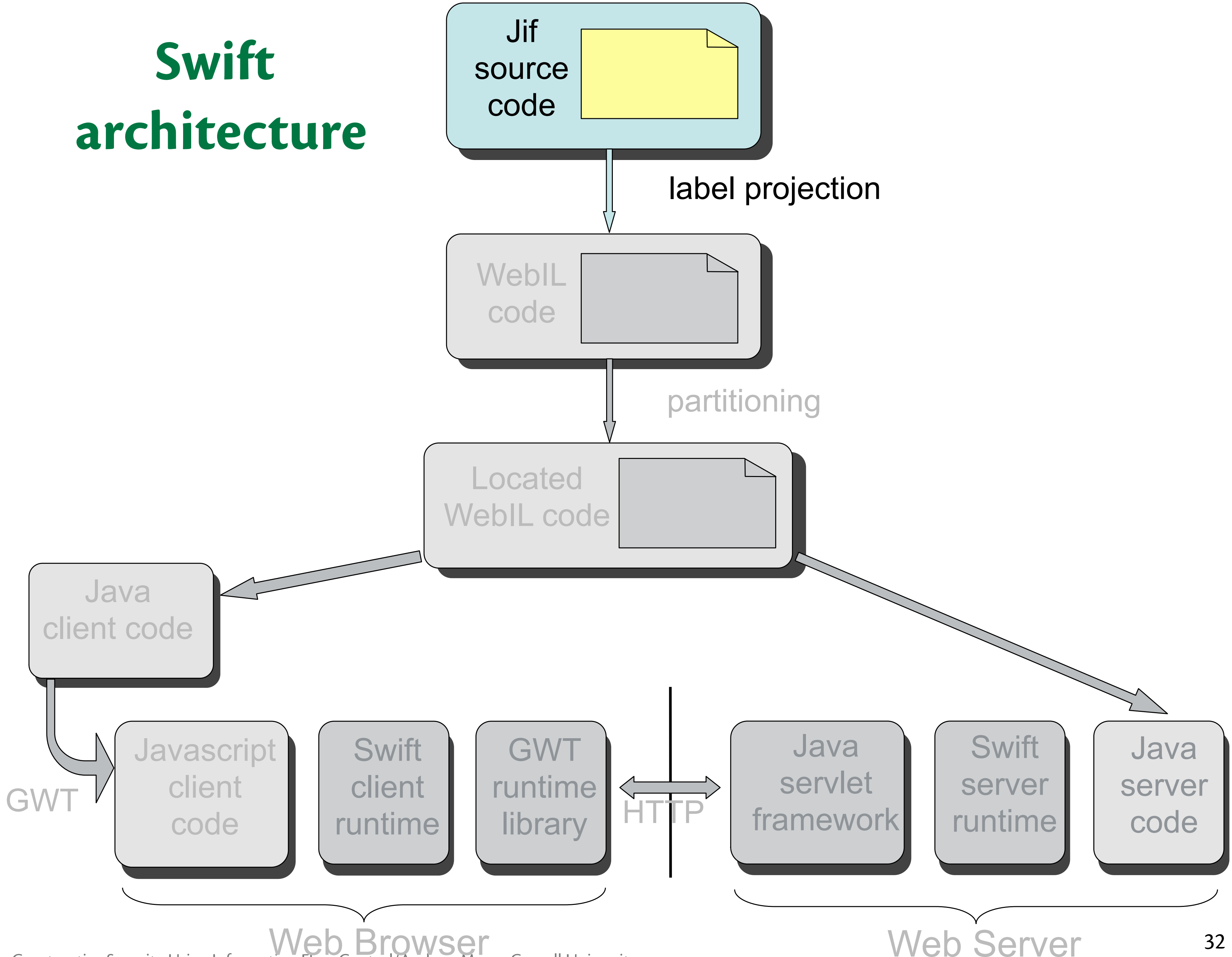
client may learn if guess is correct: declassify (requires authority of `server`)

violation of **robust declassification**: client can affect information release

Swift architecture



Swift architecture



Labels→placement constraints

{Alice→Bob, Dave}

{Eve←Chuck, Alice}

{}

{Alice→Bob, Dave}

{Alice→Bob, Dave; w}

{Alice→Bob, Dave}

{Dave→Bob, Heather}

{George→Bob, Dave; Fiona→Bob; George←Alice,Dave}

{p→Bob, q; n}

{Irina→Bob; Heather←Dave,Bob,Irina}

{Alice→Bob; Alice←Bob}

{p←p}

{Fiona→Bob,Eve,Alice; Bob←Fiona}

{Chuck→Alice,Bob;Alice←Chuck}

{Chuck←Chuck, Alice}

{Chuck←Chuck, Alice}

{Chuck←Bob, Alice}

{*l}

{x}

(low integrity) client can write

(high integrity) client cannot write

$\{\top \leftarrow \textbf{client}\} \sqsubseteq L$ $\{\top \leftarrow \textbf{client}\} \not\sqsubseteq L$

client can read (low confidentiality) $L \sqsubseteq \{\top \to \textbf{client}\}$

client cannot read (high confidentiality) $L \not\sqsubseteq \{\top \to \textbf{client}\}$

	$\{\top \leftarrow \textbf{client}\} \sqsubseteq L$	$\{\top \leftarrow \textbf{client}\} \not\sqsubseteq L$
$L \sqsubseteq \{\top \to \textbf{client}\}$	client or server S?C?	server and maybe client Sh C?
$L \not\sqsubseteq \{\top \to \textbf{client}\}$	server only S	server only Sh

Labels→placement constraints

	(low integrity) client can write	(high integrity) client cannot write
client can read (low confidentiality)	S?C?	ShC?
client cannot read (high confidentiality)	S	Sh

More placement constraints

security constraints

S?C?

ShC?

architectural constraints

S

SC

Sh

database library calls

UI widget calls

WebIL

```
Sh:      int secret;
ShC?:    int tries;
         ...
         void makeGuess (int guess)
         {
ShC?:      if (guess >= 1 && guess <= 10) {
Sh:          boolean correct = (guess == secret);
Sh:          if (tries > 0 && correct) {
S?C?:          finishApp("You win $500!");
             } else {
ShC?:          tries--;
S?C?:          if (tries > 0)
C:               message.setText("Try again");
S?C?:          else finishApp("Game over");
             }
           } else {
C:           message.setText("Out of range:" + guess);
           }
         }
```

Label forces comparison on server

calls to UI methods on client

Some placements undetermined

WebIL with placements

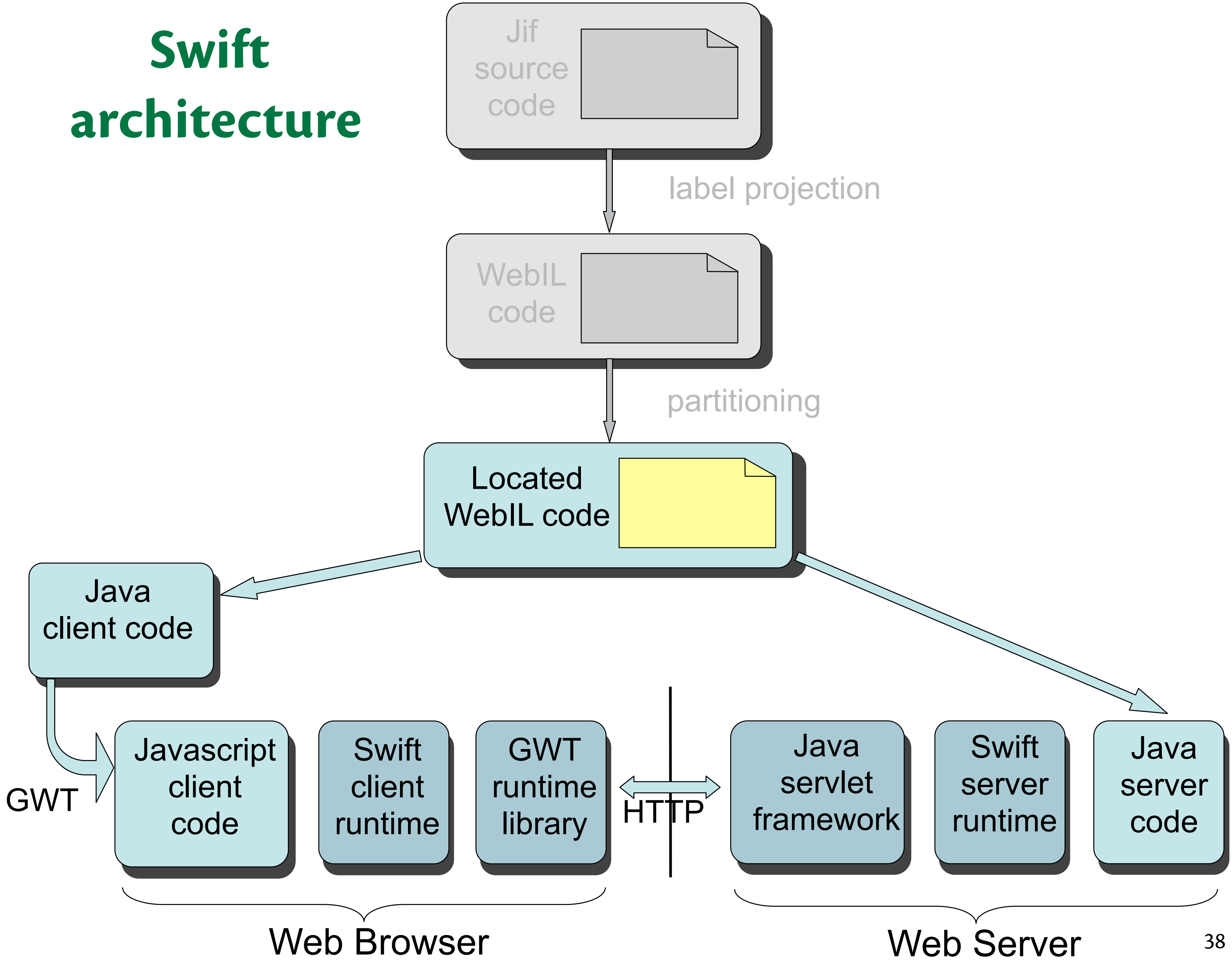
```
Sh:    int secret;
ShC:   int tries;
       ...
       void makeGuess (int guess)
       {
ShC:     if (guess >= 1 && guess <= 10) {
Sh:        boolean correct = guess == secret;
Sh:        if (tries > 0 && correct) {
C:           finishApp("You win $500!");
           } else {
ShC:         tries--;
C:           if (tries > 0)
C:             message.setText("Try again");
C:           else finishApp("Game over");
           }
         } else {
C:         message.setText("Out of range:" + guess);
         }
       }
```

Input validation code replicated

Each statement/field annotated with one of {C, S, SC, Sh, ShC}

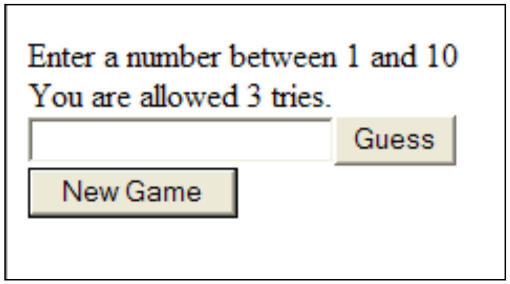
Annotations chosen to minimize network messages using min-cut algorithm.

Swift architecture

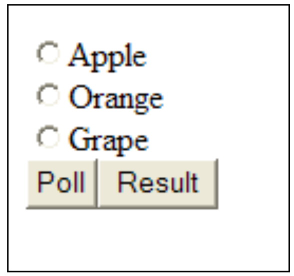


Evaluation: functionality

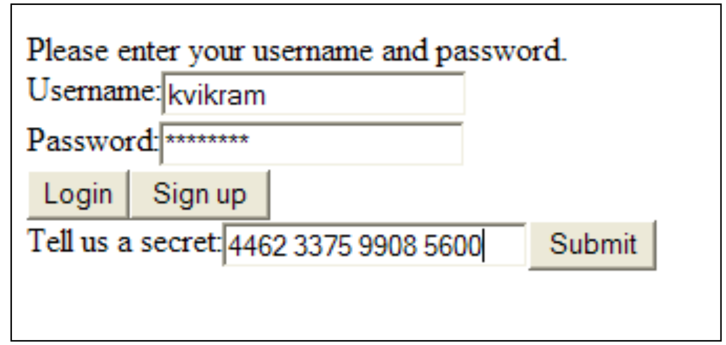
Guess-the-Number
142 lines



Poll
113 lines

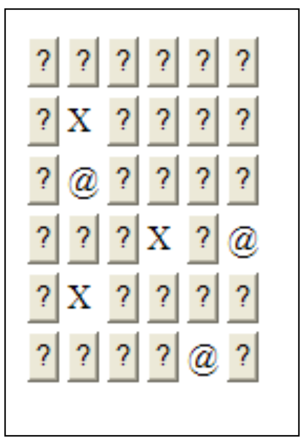


Secret Keeper
324 lines

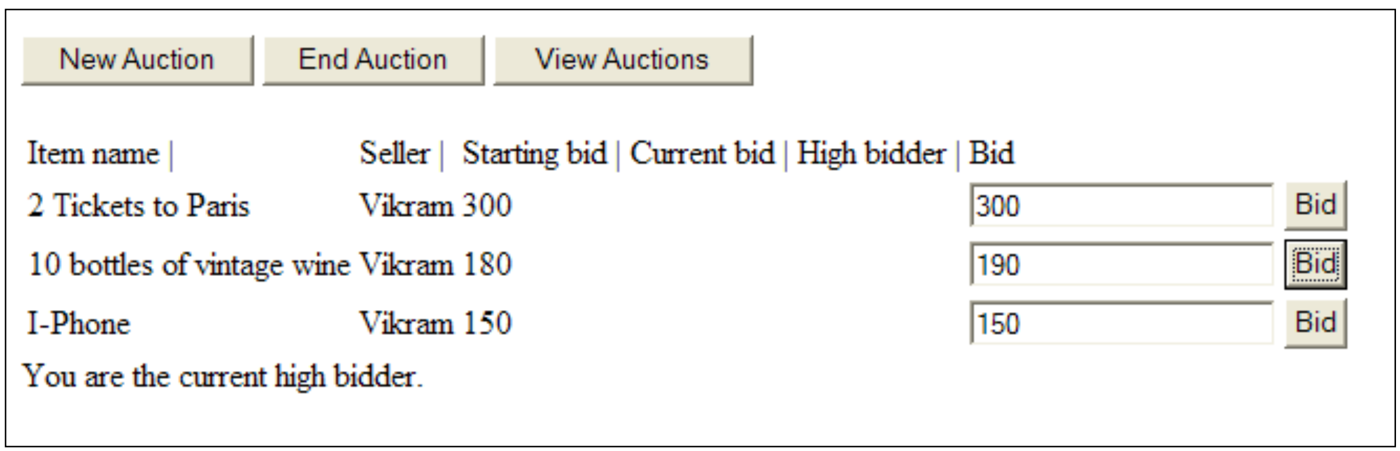


Shop
1094 lines

Treasure Hunt
92 lines



Auction
502 lines



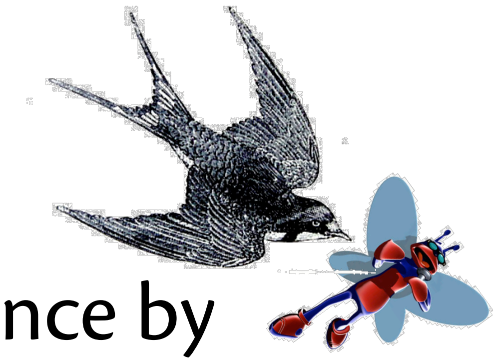
Evaluation: network messages

Example	Task	Actual		Optimal	
		Server→Client	Client→Server	Server→Client	Client→Server
Guess-the-Number	guessing a number	1	2	1	1
Shop	adding an item	0	0	0	0
Poll	casting a vote	1	1	0	1
Secret Keeper	viewing the secret	1	1	1	1
Treasure Hunt	exploring a cell	1	2	1	1
Auction	bidding	1	1	1	1

Related work

- Unified web programming models
  - Links [CLWY06]
  - Hop [SGL06]
  - Hilda [YGQDGS07,YSRG 06]
- Web application security
  - Static analysis [HYHTLK 04, LL05, X06, XA06, JKK06]
  - Information flow via dynamic taint tracking [HO05, NGGE05, XBS06, CVM07]
- Security by construction
  - Jif/split [ZZNM02, ZCMZ03] and provably sound impls of partitioning [FR08, FGR09]
  - Fairplay [MNPS04]
  - SMCL [NS07]

Swift summary

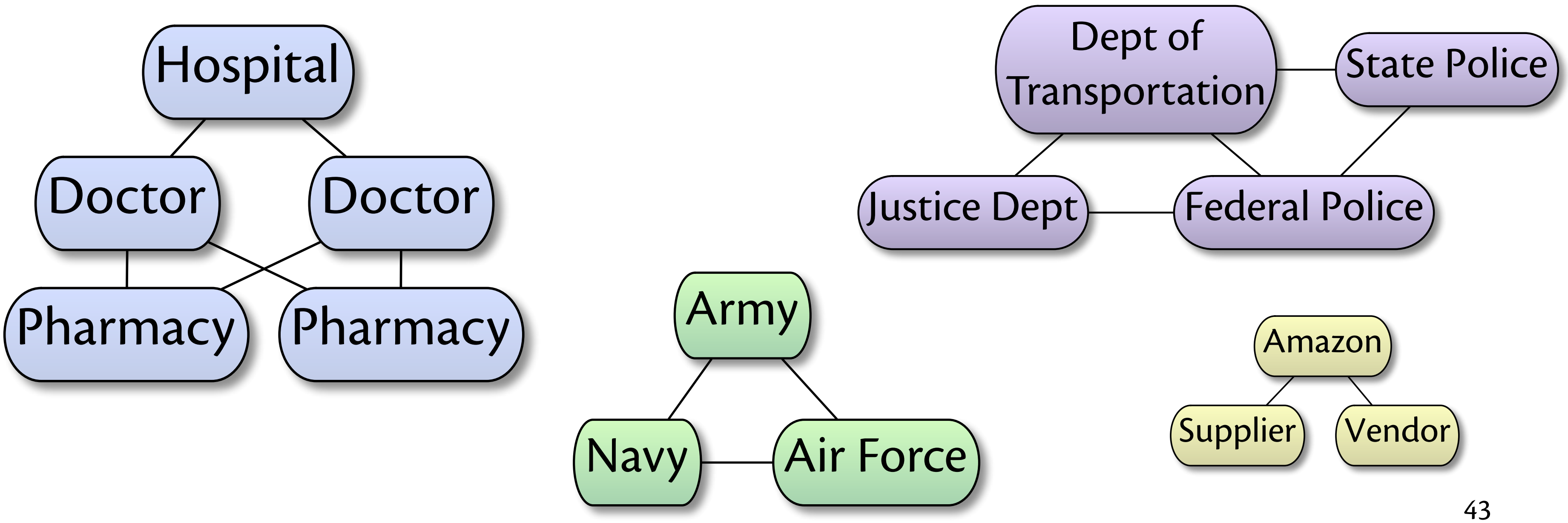


- Web applications with security assurance by construction
  - cleaner, higher-level programming model
  - enabled by declarative security annotations
  - automated enforcement ⇒ greater security assurance
  - security-constrained optimization

- What about more general distributed computation?

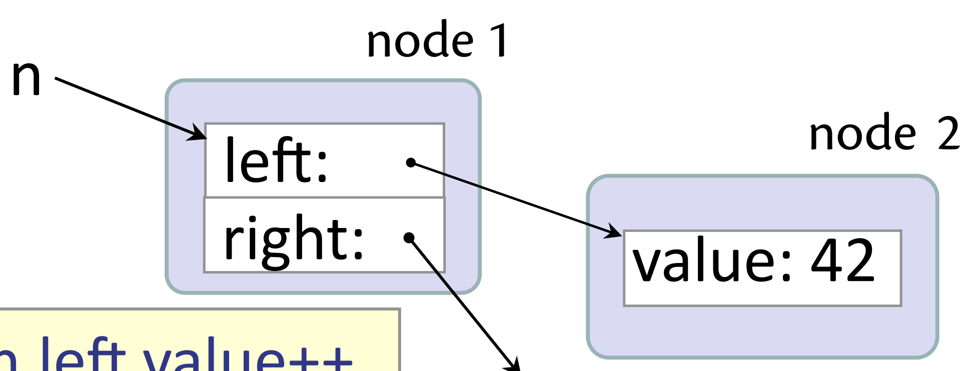
Decentralized sharing?

- *Federated systems* integrate data and computation across administrative boundaries
  - can add functionality, increase automation
  - Web is federated but not very programmable
  - Need **security** and **consistency**



Fabric : a system and a language [SOSP 09]

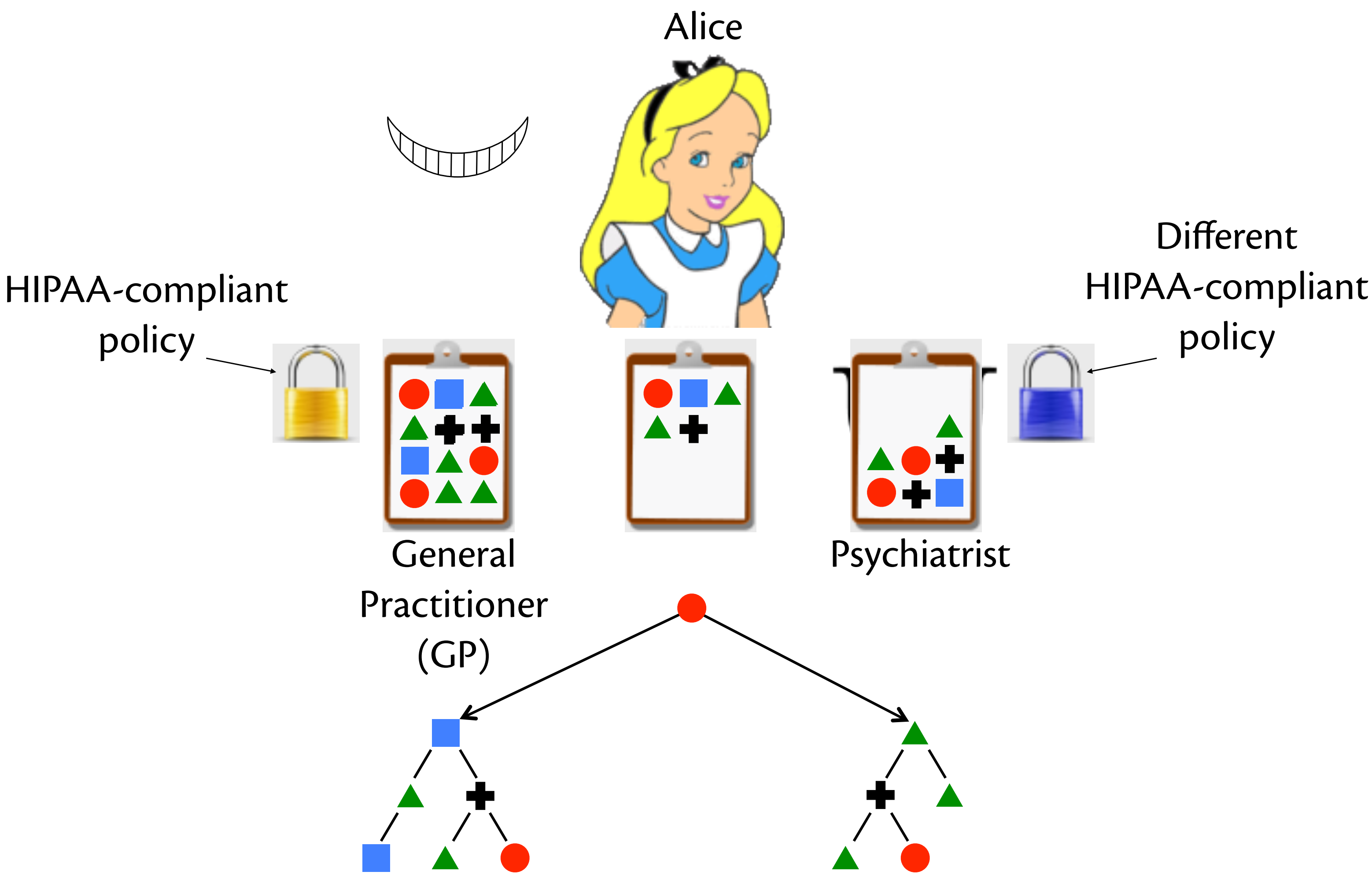
- Goal: a undergraduate can write secure, reliable programs for the Internet Computer
- All information (persistent or otherwise) looks like an ordinary program object
- Objects connected by references
  - Any object can be referenced uniformly from anywhere
  - References look like ordinary object pointers but can cross nodes and trust domains



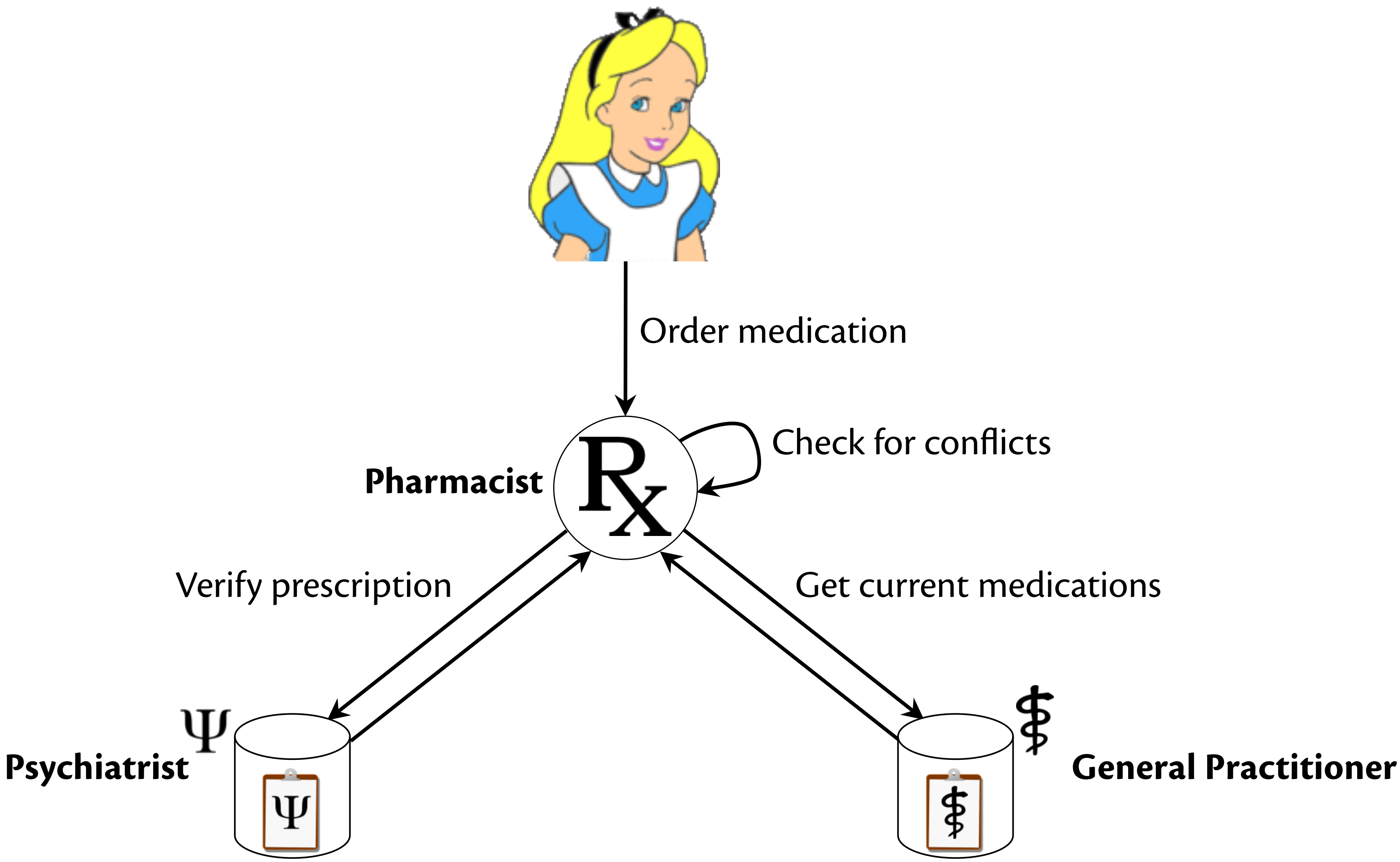
n.left.value++

Compiler and runtime enforce security and consistency despite distrust

Fabric enables federated sharing



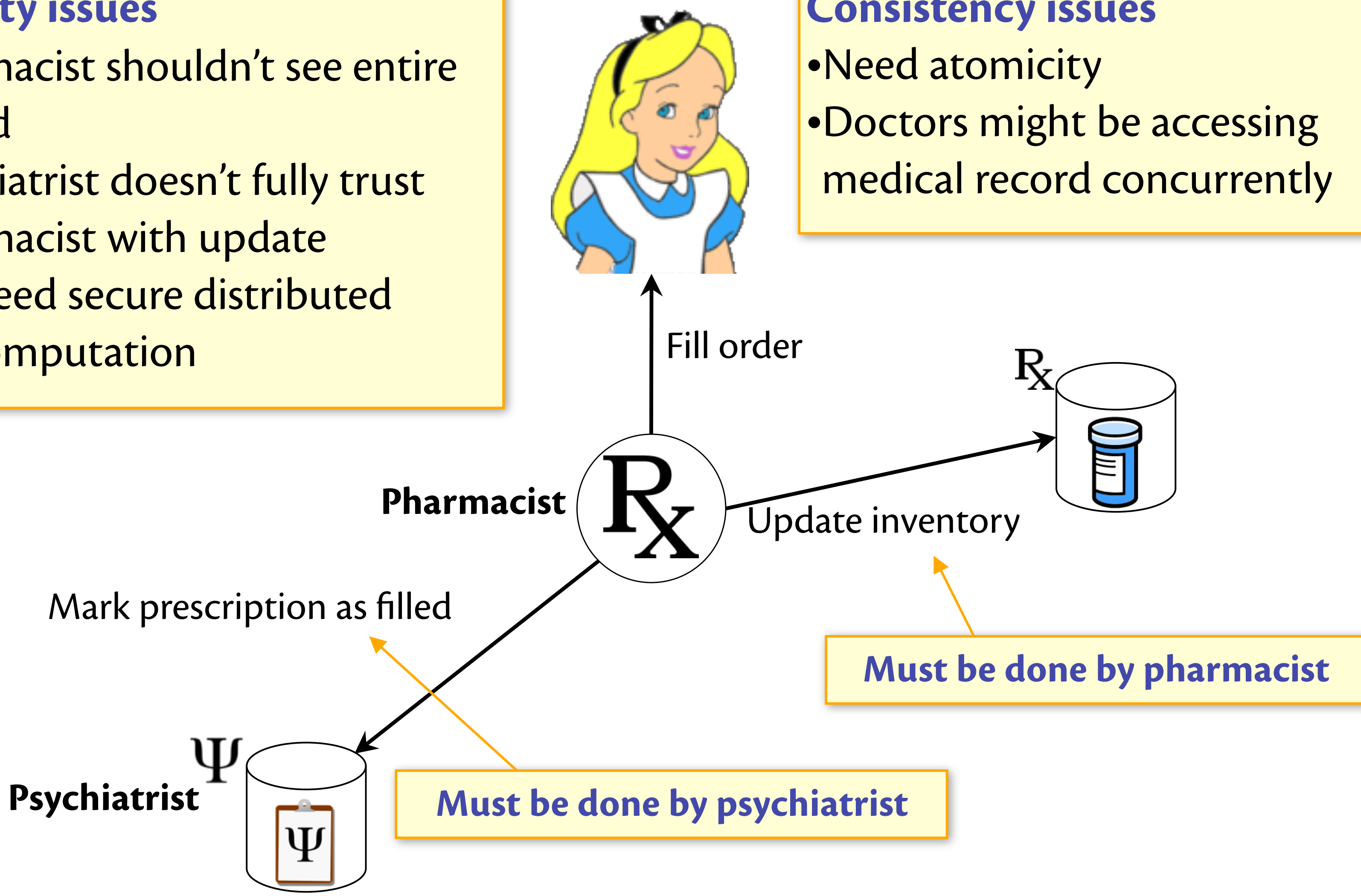
Example: Filling a prescription



Example: Filling a prescription

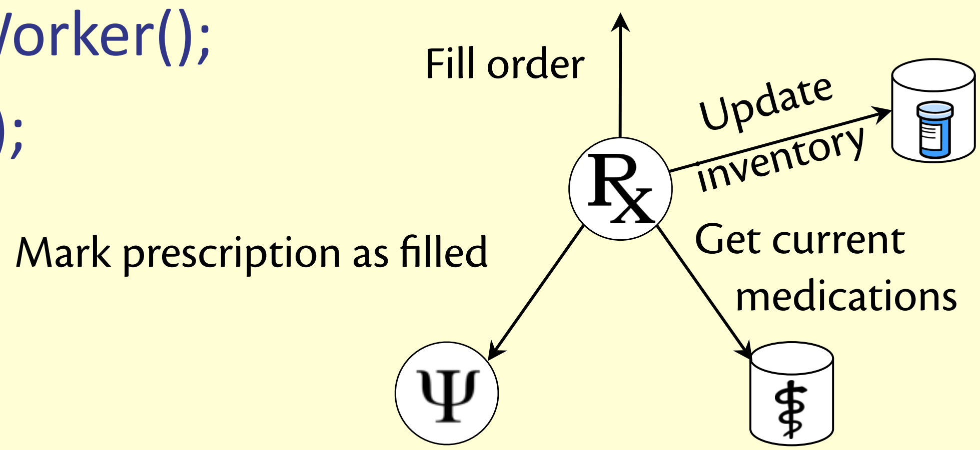
Security issues
- Pharmacist shouldn't see entire record
- Psychiatrist doesn't fully trust pharmacist with update
  - Need secure distributed computation

Consistency issues
- Need atomicity
- Doctors might be accessing medical record concurrently



Pharmacy example in Fabric

```
Order orderMed(PatRec psyRec, PatRec gpRec, Prescription p) {
  atomic {
    if (!psyRec.hasPrescription(p)) return Order.INVALID;
    if (isDangerous(p, gpRec.getMeds())) return Order.DANGER;

    Worker psy = psyRec.getWorker();
    psyRec.markFilled@psy(p);
    updateInventory(p);
    return Order.fill(p);
  }
}
```



Fabric: a high-level language

```
Order orderMed(PatRec psyRec, PatRec gpRec, Prescription p) {
  atomic {
    if (!psyRec.hasPrescription(p)) return Order.INVALID;
    if (isDangerous(p, gpRec.getMeds())) return Order.DANGER;

    Worker psy = psyRec.getWorker();
    psyRec.markFilled@psy(p);
    updateInventory(p);
    return Order.fill(p);
  }
}
```

Java with:

- Remote calls
- Nested transactions (atomic blocks)
- Label annotations for security (elided)

Fabric: a high-level language

```
Order orderMed(PatRec psyRec, PatRec gpRec, Prescription p) {
  atomic {
    if (!psyRec.hasPrescription(p)) return Order.INVALID;
    if (isDangerous(p, gpRec.getMeds())) return Order.DANGER;

    Worker psy = psyRec.getWorker();
    psyRec.markFilled@psy(p);
    updateInventory(p);
    return Order.fill(p);
  }
}
```

- All objects accessed uniformly regardless of location
- Objects fetched transparently as needed
- Remote calls are explicit

Remote calls

```
Order orderMed(PatRec psyRec, PatRec gpRec, Prescription p) {
  atomic {
    if (!psyRec.hasPrescription(p)) return Order.INVALID;
    if (isDangerous(p, gpRec.getMeds())) return Order.DANGER;


    Worker psy = psyRec.getWorker();
    psyRec.markFilled@psy(p);
    updateInventory(p);
    return Order.fill(p);
  }
}
```

Remote call — pharmacist runs code at psychiatrist's node

Federated transactions

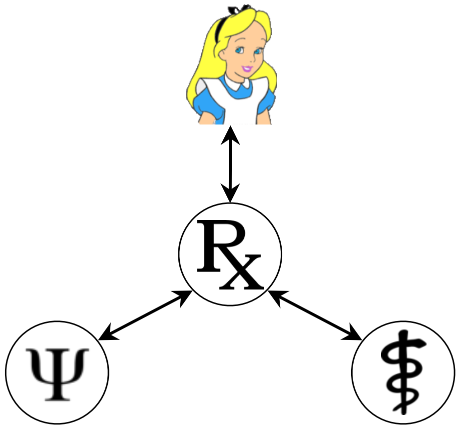
```
Order orderMed(Pat                              Prescription p) {
  atomic {
    if (!psyRec.hasPrescription(p)) return Order.INVALID;
    if (isDangerous(p, gpRec.getMeds())) return Order.DANGER;

    Worker psy = psyRec.getWorker()
    psyRec.markFilled@psy(p);
    updateInventory(p);
    return Order.fill(p);
  }
}
```

Federated transaction — spans multiple nodes & trust domains

Remote call — pharmacist runs method at psychiatrist's node

Fabric security model

- Decentralized system
  - *anyone* can join
  - No centralized enforcement
- **Decentralized security principle:**
  - **You can't be hurt by what you don't trust**



Security labels in Fabric

Confidentiality:	Alice → Bob	Alice permits Bob to *learn*
Integrity:	Alice ← Bob	Alice permits Bob to *affect*

```
class Prescription {
  Drug{Psy→A_pharm ; Psy←Psy} drug;
  Dosage{Psy→A_pharm ; Psy←Psy} dosage;
… }
```

- Compiler and runtime together ensure policies are not violated by any information flows in system.

Trust management

- Fabric principals are objects

```
class Principal {
  boolean delegatesTo(principal p);
  void addDelegatesTo(principal p) where caller (this);
  ...
}
```

Determines whether *p* acts for this principal

Must have authority of this principal to call

- Explicit trust delegation via method calls

```
// Assert "Alice acts-for Bob"
bob.addDelegatesTo(alice)
```

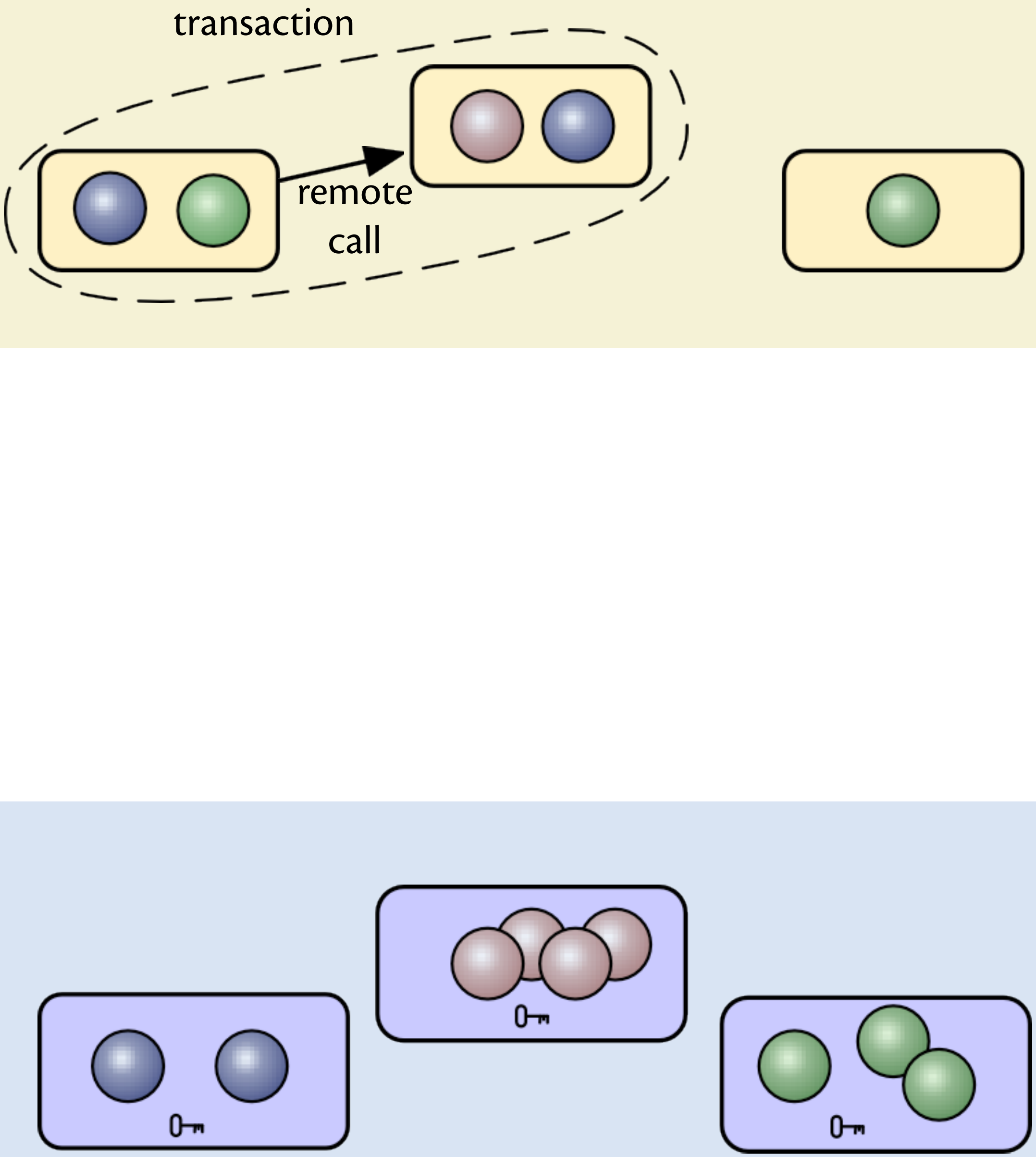
  – Compiler and run-time ensure that caller has proper authority

Fabric abstraction

- Fabric language combines:
  - Information flow policy annotations
  - Remote calls
  - (Optimistic) nested atomic transactions
- Fabric system is a decentralized platform for secure, consistent sharing of information and computation
  - Nodes join freely
  - No central control over security

How to build a system that implements this abstraction?

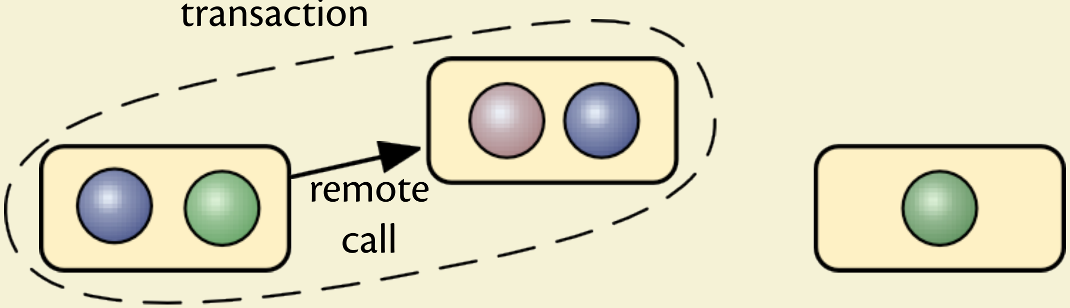
Fabric Architecture



Worker nodes (Workers)

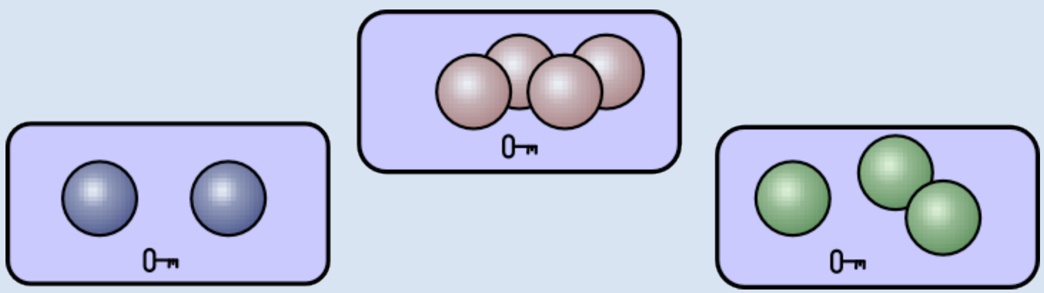
Storage nodes (Stores)

Fabric Architecture



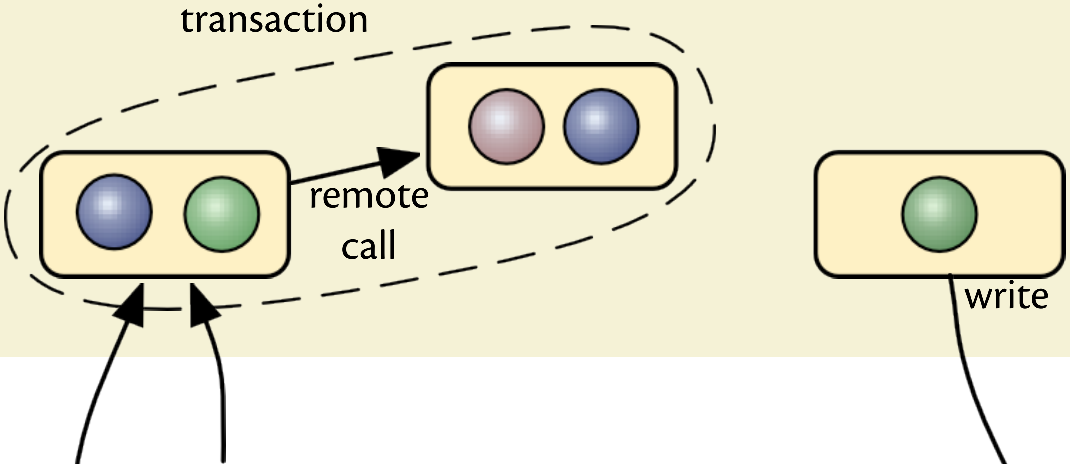
Worker nodes

(Workers)

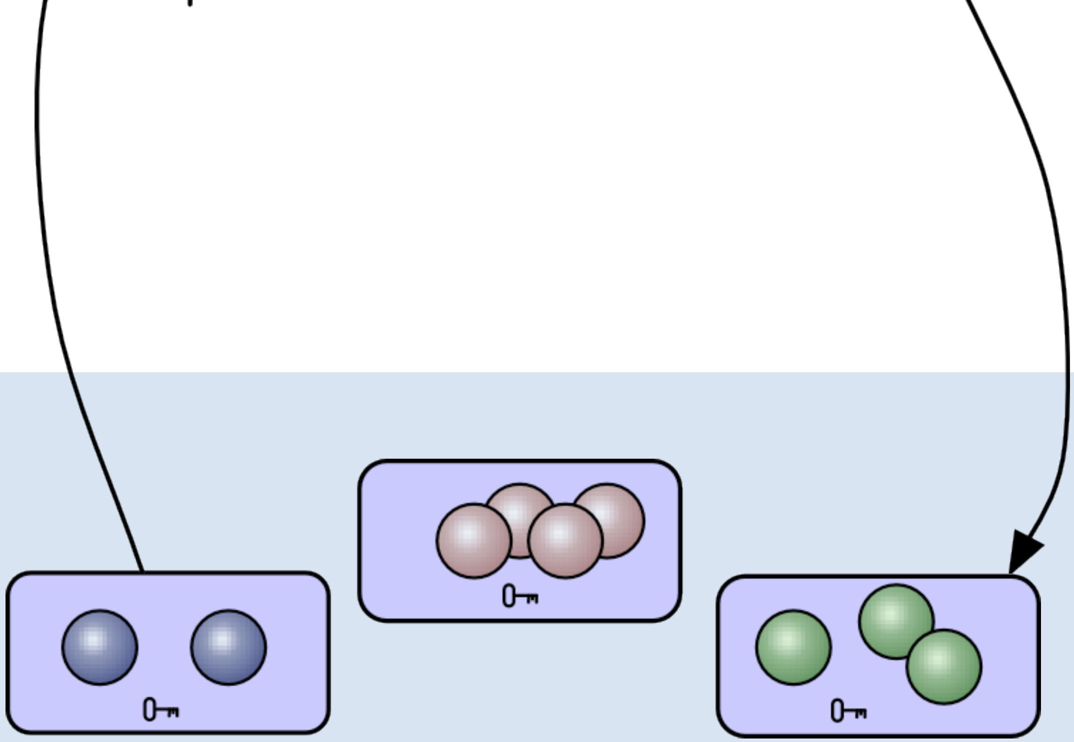


- **Storage nodes** securely store persistent objects
- Each object specifies its own security policy, enforced by store

Fabric Architecture

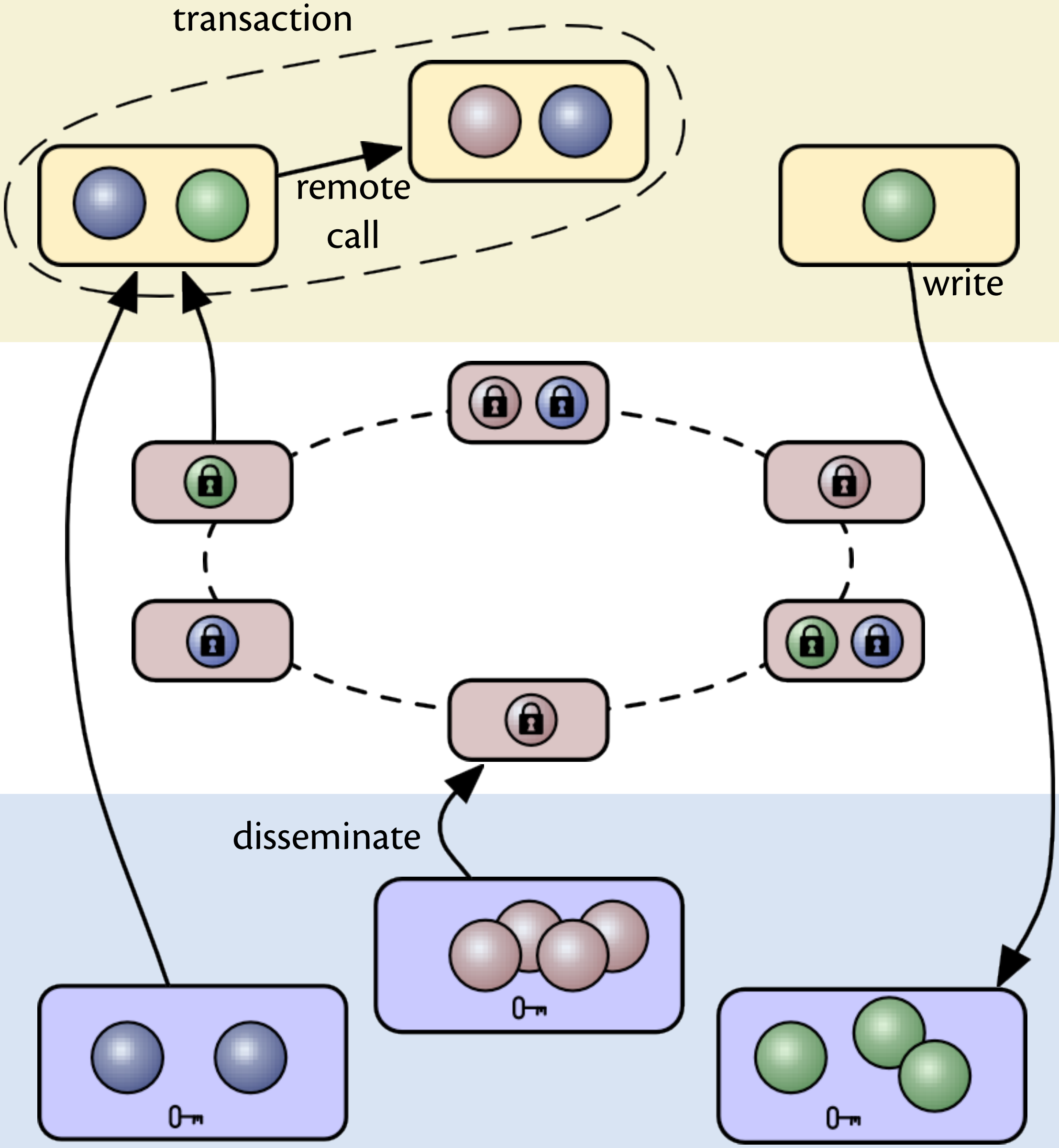


- **Worker nodes** compute on cached objects
- Computation may be distributed across workers in **federated transactions**



- **Storage nodes** securely store persistent objects
- Each object specifies its own security policy, enforced by store

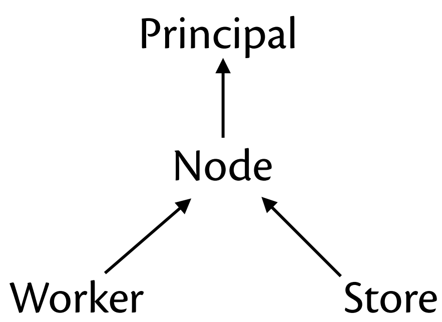
Fabric Architecture



- **Worker nodes** compute on cached objects
- Computation may be distributed across workers in **federated transactions**
- **Dissemination nodes** cache signed, encrypted objects in peer-to-peer distribution network for high availability
- **Storage nodes** securely store persistent objects
- Each object specifies its own security policy, enforced by store

Fabric run-time system

- Nodes are principals in Fabric language



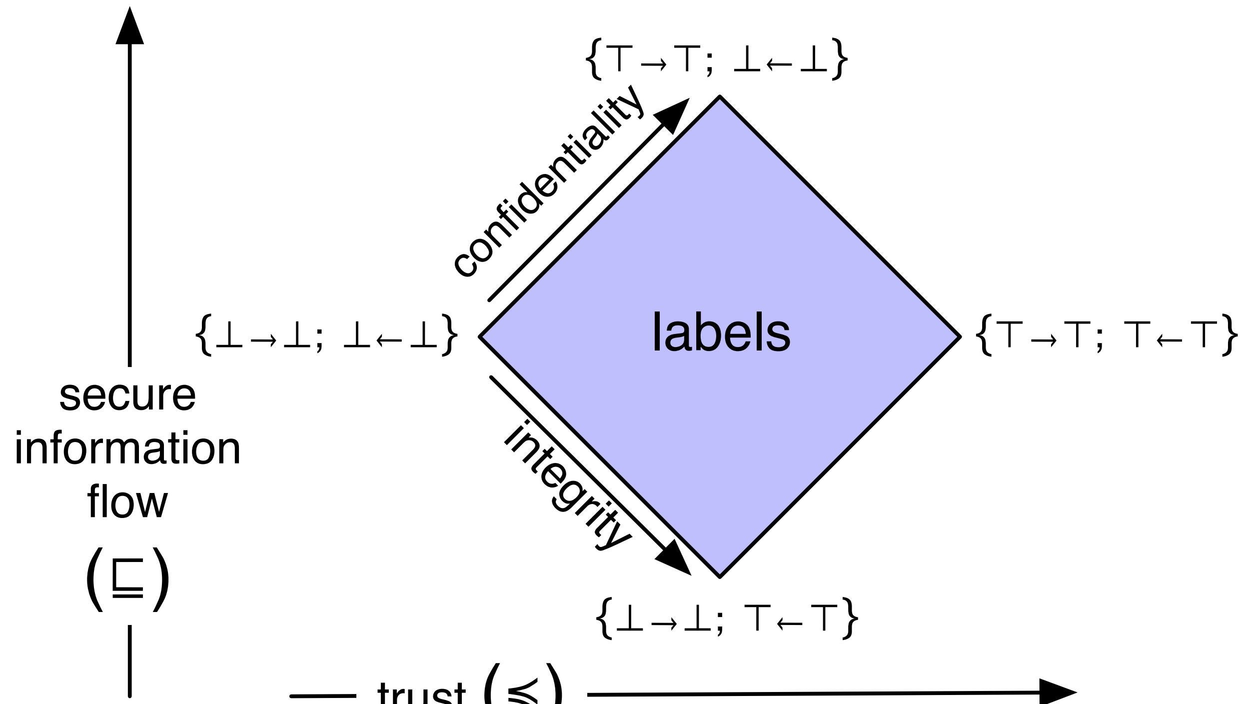
- Root of trust: X.509 certificates bind hostnames to node principal objects
  - Store getStore(String hostname) checks certificate
  - Nodes act for principals stored at them.

Secure data placement

- Placing objects with label L securely: is node n trusted to enforce label L?

`new Foo@n(...)` → **Static check** $\{\top\leftarrow n;\ \top\rightarrow n\} \succcurlyeq L$

- **Trust ordering** $\succcurlyeq$ on labels lifts principal acts-for ordering $\succcurlyeq$ to relate information flow policies.



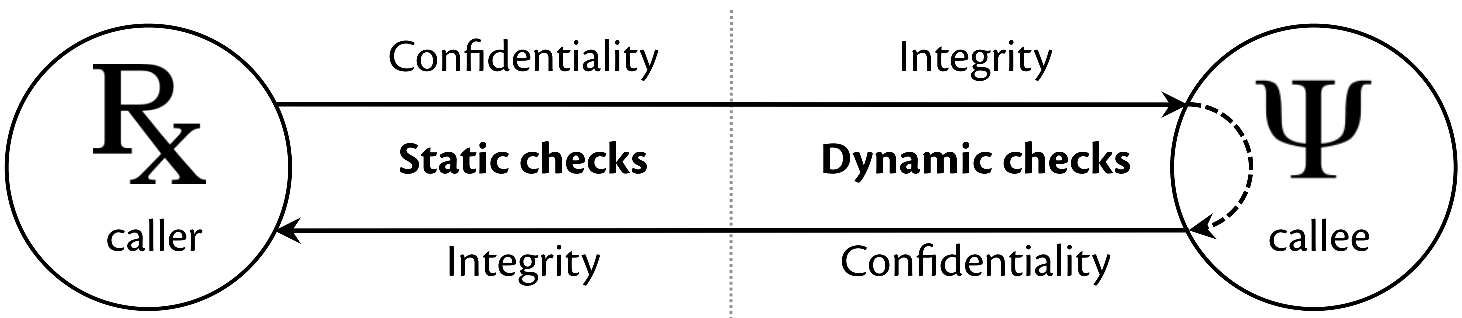
Secure remote calls

Is callee trusted to see call?

- Call itself might reveal private information
- Arguments might be private

Is caller trusted to make call?

- Caller might lack sufficient authority to make call
- Method arguments might have been tampered with by caller



Is callee trusted to execute call?

- Result might have been tampered with by callee

Is caller trusted to see result?

- Call result might reveal private information

Result: secure information flow enforced end-to-end across network

and more mechanisms...

- Writer maps for secure propagation of updates
- Automatic 'push' of updated objects to dissemination layer
- In-memory caching of object groups at store
- Object-group clustering and prefetching
- Caching and incrementally updating acts-for relationships
- Secure distributed transaction logs
- Hierarchical two-phase commit protocol

(see the SOSP'09 paper)

Implementation

- Fabric prototype implemented in Java and Fabric
  - Total: 35 kLOC
  - Compiler translates Fabric into Java
    - 15 k-line extension to Jif compiler using Polyglot [NCM03]
  - Dissemination layer: 1.5k-line extension to FreePastry
    - Popularity-based replication (à la Beehive [RS04])
  - Store uses BDB as backing store

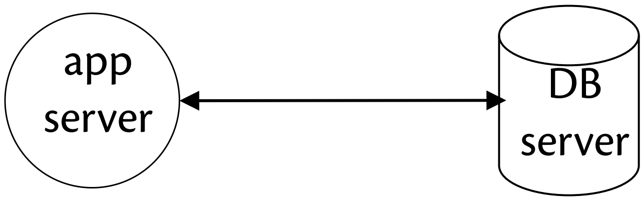
Object overheads

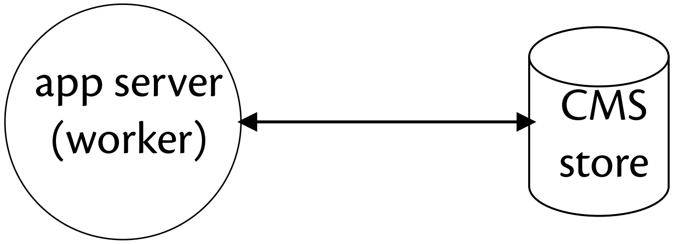
- Extra overhead on object accesses at worker
  - Run-time label checking
  - Logging reads and writes
  - Cache management (introduces indirection)
  - Transaction commit
- Overhead at store for reads and commits
- Ported non-trivial web application to evaluate performance: a course management system.

CMS experiment

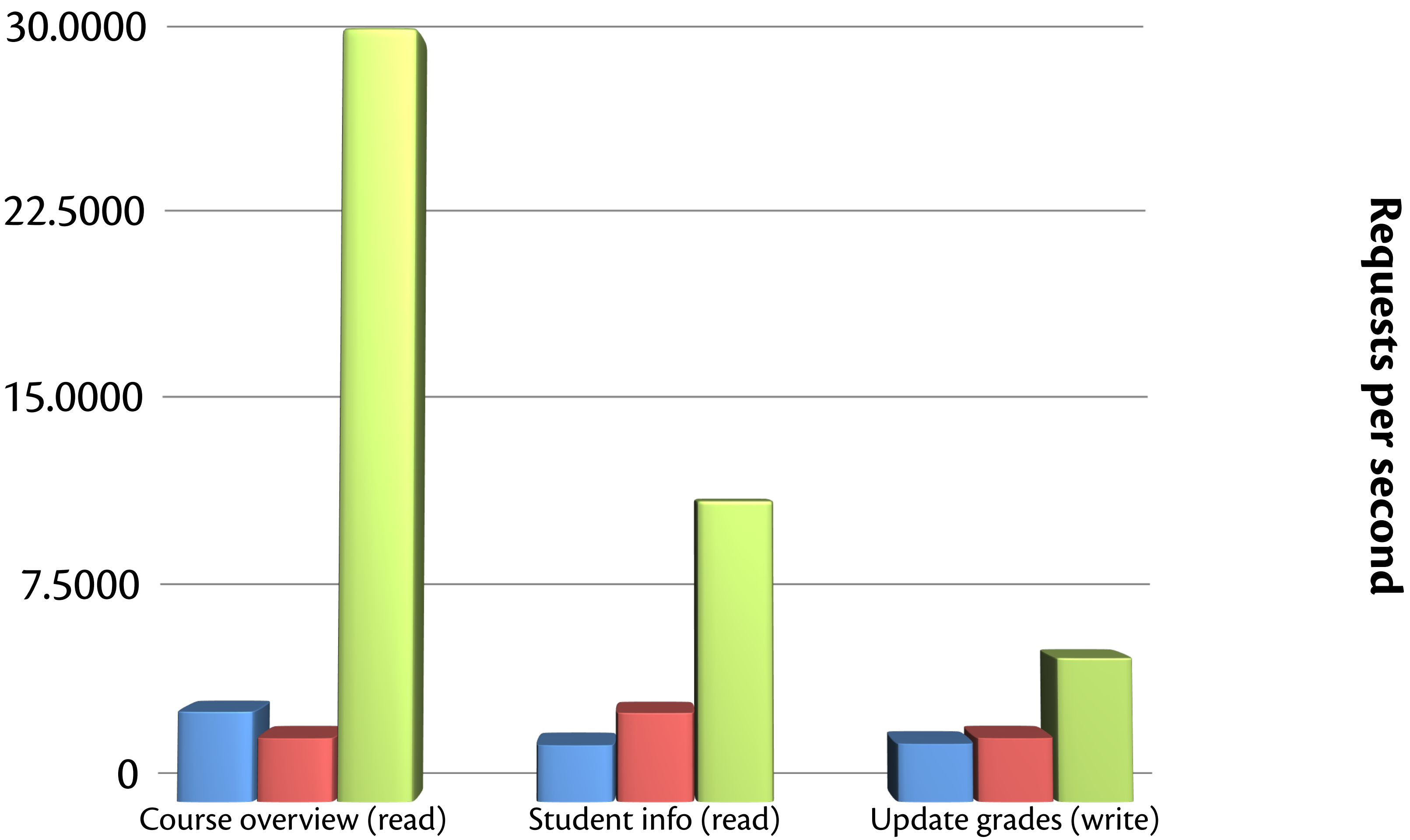
- CMS has been used at Cornell since 2004
  - Over 2000 students in over 40 courses
- Two prior implementations using SQL database:
  - J2EE/EJB2.0 (production system) [BCCDGGGLPRRYACGMS05]
    - 54k-line web app with hand-written SQL
    - Oracle database
  - Hilda [YGG+07]
    - High-level language for data-driven web apps
- Fabric implementation:

  3k lines → 740 lines





Performance results



- EJB, Hilda: DB server must be contacted frequently.
- Fabric: persistent objects can be cached at app server.

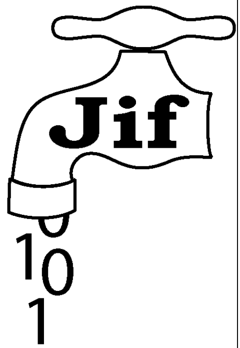
Related work: Fabric

Category	Examples	Fabric adds:
Federated object store	OceanStore/Pond	• Transactions • Security policies
Secure distributed storage systems	Boxwood, CFS, Past	• Fine-grained security • High-level programming
Distributed object systems	Gemstone, Mneme, ObjectStore, Sinfonia, Thor	• Security enforcement • Multi-worker transactions with distrust
Distributed computation/ RPC	Argus, Avalon, CORBA, Emerald, Live Objects, Network Objects	• Single-system view of persistent data • Strong security enforcement
Distributed information flow systems	DStar, Jif/split, Swift	• Consistency for shared persistent data

No prior system has provided the security and expressiveness of Fabric.

Constructive security×3

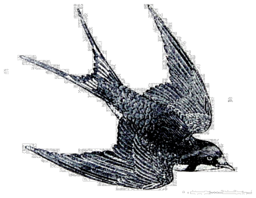
- **Jif**
  - adding information flow policies to a real programming language
  - compiler supports programmer reasoning about security
- **Swift**
  - automatically, securely partitioning web applications
- **Fabric**
  - a general, high-level abstraction for secure, consistent, federated computing
- A truly secure Internet Computer requires raising the level of abstraction even higher
  - Decentralization and federation (ala Fabric) + automatic mapping of code and data (ala Swift)
  - Many challenges: mobile code; dynamic, adaptive partitioning; efficient, secure data management; richer compositional policies; formal security proofs; consistency policies; synthesizing more crypto protocols



Conclusions

Information flow policies enable a constructive approach to security:

- stronger, end-to-end, compositional security
- higher-level, more abstract programming model
- opportunities for greater efficiency and automatic optimization



Acknowledgments

- Steve Chong
- Jed Liu
- Nate Nystrom
- Xin Qi
- Mike George
- K. Vikram
- Steve Zdancewic
- Lantian Zheng
- Xin Zheng

Jif

Fabric

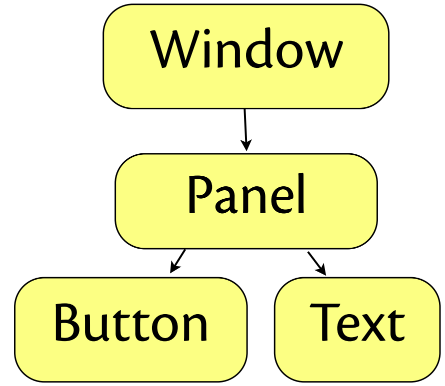
Additional material

The following slides were not used in the talk but may help answer questions.

Covert channels

- Confidentiality depends on adversary not learning things from observations
- Information flow control prevents learning from observations at language level of abstraction (exception: termination vs. nontermination)
- Lower-level observations might still leak information:
  - Time and power
  - Size, existence, source, destination of network messages
  - Nondeterministic choices: addresses, interleavings, ...
  - Lower-level protocol message contents
- Run-time mechanisms exist for mitigating them

GUI interfaces



- Swift is a GUI toolkit similar to Swing (Java)
  – Layout is dynamic and user events are handled securely
- Information flow tracked through GUI widgets
  – `Out` and `In` labels bound information flowing up and down through hierarchy.

```
class Widget[label Out, label In] { ... }
class Panel[label Out, label In]
 extends Widget[Out,In] {
  void addChild{Out}(label wOut, label wIn, Widget[wOut,wIn]{Out} w)
    where {*wOut} <= Out, {In;w} <= {*wIn};
}
class ClickableWidget[label Out, label In]
 extends Widget[Out,In] {
  void addListener{In}(ClickListener[Out,In]{In
class Button[label Out, label In] extends ClickableWidget[Out,In] {
  String{Out} getText();
  void setText{Out}(String{Out} text);
}
interface ClickListener[label Out, label In] {
  void onClick{In}(Widget[Out, In]{In} b);
}
```

Classes can be parameterized on labels and principals

Child widget must agree statically with parent—bad hierarchies ruled out at compile time.