STORM

Refinement Types for Secure Web Applications

https://storm-framework.github.io

Nico Lehmann *Rose Kunkel* *Jordan Brown* *Niki Vazou*

Jean Yang *Nadia Polikarpova* *Deian Stefan* ***Ranjit Jhala***

University of California, San Diego

Securing Web Applications

OWASP “Top 10”

Top 10 Web Application Security Risks

A1:2017-Injection: Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

A2:2017-Broken Authentication: Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities temporarily or permanently.

A3:2017-Sensitive Data Exposure: Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

A4:2017-XML External Entities (XXE): Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.

A5:2017-Broken Access Control: Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users’ accounts, view sensitive files, modify other users’ data, change access rights, etc.

A6:2017-Security Misconfiguration: Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.

A7:2017-Cross-Site Scripting XSS: XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

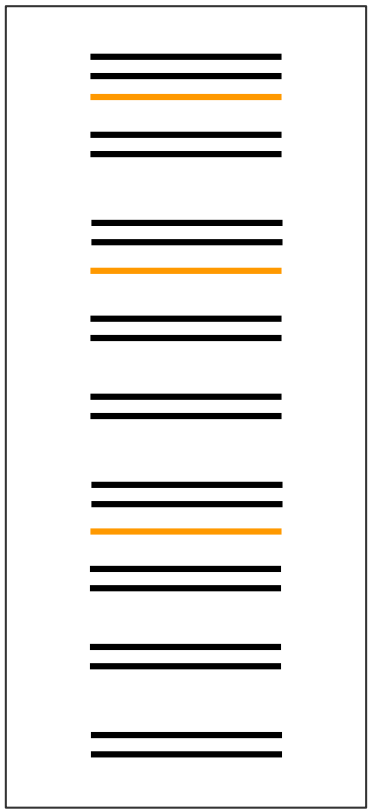
Security Mixed Into “Business Logic”

```
if ($Me->can_view_review($prow, $rrow)) {
    $rname = Ht::link($rname, $prow->reviewurl(["r" => $rrow->reviewId]));
}
echo $rname, ': ', $namex,
    '</div><div class="f-h"><ul class="x mb-0">';
echo '<li>requested';
if ($rrow->timeRequested) {
    echo ' ', $Conf->unparse_time_relative((int) $rrow->timeRequested);
}
if ($rrow->requestedBy == $Me->contactId) {
    echo " by you";
} else if ($Me->can_view_review_requester($prow, $rrow)) {
    echo " by ", $Me->reviewer_html_for($rrow->requestedBy);
}
echo '</li>';
if ($rrow->reviewStatus === ReviewInfo::RS_ACCEPTED) {
    echo '<li>accepted';
    if ($req[1]) {
        echo ' ', $Conf->unparse_time_relative($req[1]);
    }
    echo '</li>';
}
```

```
$myrrow = $apprrow1 = $apprrow2 = null;
$admin = $this->user->can_administer($this->prow);
foreach ($this->prow->reviews_as_display() as $rrow) {
    if ($this->user->can_view_review($this->prow, $rrow)) {
        if ($rrow->contactId === $this->user->contactId
            || (!$myrrow && $this->user->is_my_review($rrow))) {
            $myrrow = $rrow;
        } else if ($rrow->reviewStatus === ReviewInfo::RS_DELIVERED
                   && !$apprrow1
                   && $rrow->requestedBy === $this->user->contactXid) {
            $apprrow1 = $rrow;
        } else if ($rrow->reviewStatus === ReviewInfo::RS_DELIVERED
                   && !$apprrow2
                   && $admin) {
            $apprrow2 = $rrow;
        }
    }
}
```

HotCRP.com

Security Mixed Into "Business Logic"



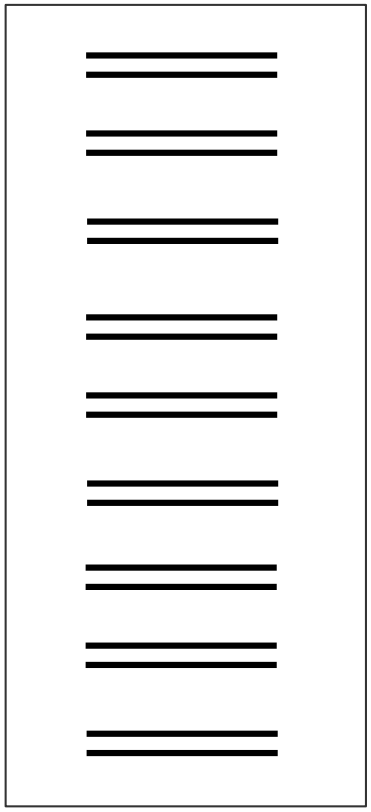
Does application enforce high-level security policy?

STORM

Separate Policy from "Business Logic"

Policy

Audit & Trust



Application

Verify!

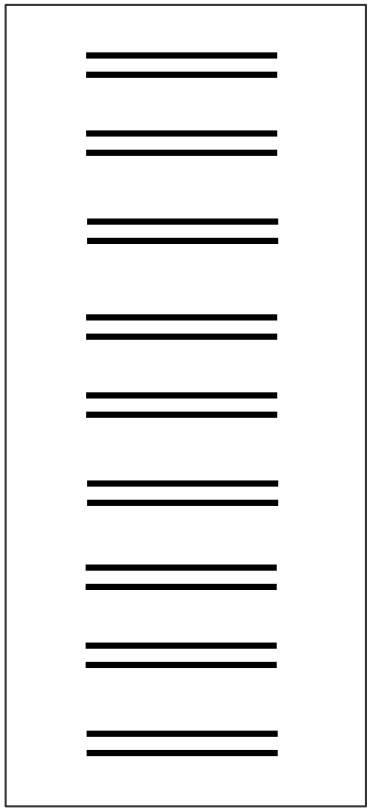
Policies are *much smaller* than application code

STORM

Security Typed ORM

Policy

Audit & Trust



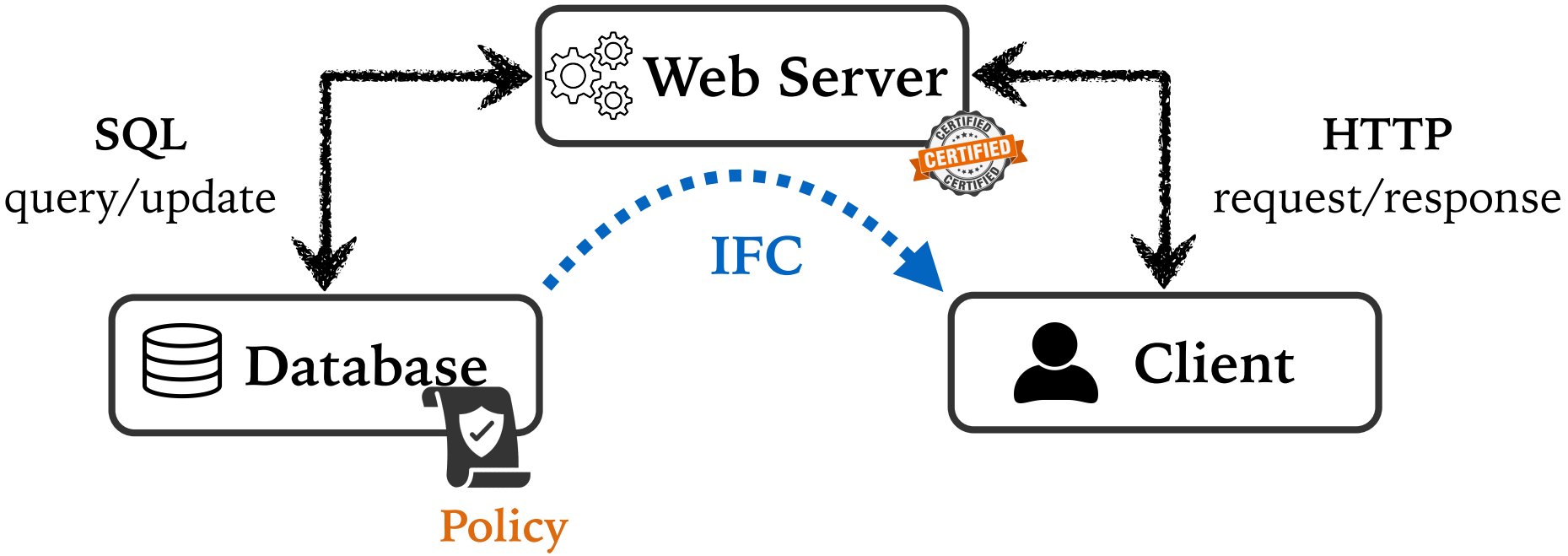
Application

Verify!

Policies are *much smaller* than application code

STORM

Security Typed ORM



Information Flow Control

By refinement type checking

I. Motivation

Why secure web applications?

II. Demonstration

How to secure apps with STORM?

III. Implementation

How does STORM enforce security?

IV. Evaluation

Can STORM secure real web-apps?

I. Motivation

Why secure web applications?

II. Demonstration

How to secure apps with STORM?

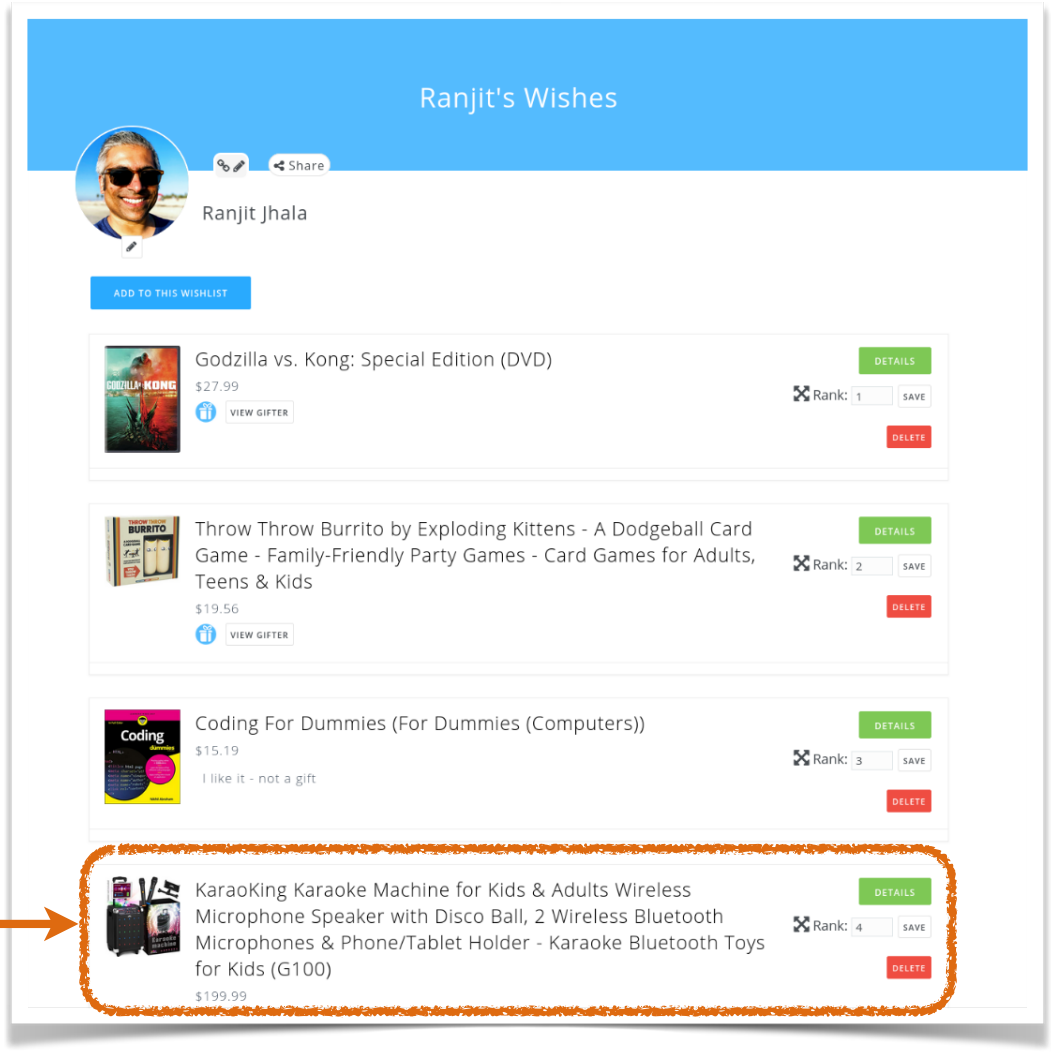
III. Implementation

How does STORM enforce security?

IV. Evaluation

Can STORM secure real web-apps?

A “Wishlist” Application



Private →

A “Wishlist” Application

Policy

Only the owner can access a private wish

A “Wishlist” Application

```
User
  name  String
  email String

Wish
  owner UserId
  title String
  level String
```

DB Schema

A “Wishlist” Application

```
User
  name  String
  email String

Wish
  owner UserId
  title String @OnlyPub
  level String
```

```
policy OnlyPub = \wish viewer ->
  wish.level == "public" ||
  wish.owner == viewer
```

DB Schema

Policy

A “Wishlist” Application

```
showWishes uid = do {
  query  <- (Owner ==. uid);
  wishes <- select query;
  titles <- project Title wishes;
  respond titles
}
```

Application Code

A “Wishlist” Application

```
showWishes uid = do {
  query  <- (Owner ==. uid);
  wishes <- select query;
  titles <- project Title wishes;
  respond titles
}
```

Application Code

A “Wishlist” Application

```
showWishes uid = do {
  query  <- (Owner ==. uid);
  wishes <- select query;
  titles <- project Title wishes;
  respond titles
}
```

Application Code

A “Wishlist” Application

```
showWishes uid = do {
  query  <- (Owner ==. uid);
  wishes <- select query;
  titles <- project Title wishes;
  respond titles
}
```

Application Code

A “Wishlist” Application

```
showWishes uid = do {
  query  <- (Owner ==. uid);
  wishes <- select query;
  titles <- project Title wishes;
  respond titles
}
```

Leak!

Application Code

A “Wishlist” Application

```
showWishes uid = do {
  query  <- (Owner ==. uid);
  wishes <- select query;
  titles <- project Title wishes;
  respond titles
}
```

Leak!

Application Code

All wishes owned by `uid`

A “Wishlist” Application

```
showWishes uid = do {
  query  <- (Owner ==. uid);
  wishes <- select query;
  titles <- project Title wishes;
  respond titles
}
```

Leak!

Application Code

Only viewable by `uid`

A "Wishlist" Application

```
showWishes uid = do {
  query  <- (Owner ==. uid);
  wishes <- select query;
  titles <- project Title wishes;
  respond titles
}
```

Leak!

Application Code

Response sent to `sessionUser != uid`

A “Wishlist” Application

```
showWishes uid = do {
  viewer <- getSessionUser();
  query  <- viewer == uid ? true : Level ==. “public”;
  wishes <- select (Owner ==. uid &&. query) ;
  titles <- project Title wishes;
  respond titles
}
```

Fix

Restrict to “public” when sessionUser != uid

A “Wishlist” Application

```
showWishes uid = do {
  viewer <- getSessionUser();
  query  <- viewer == uid ? true : Level ==. “public”;
  wishes <- select (Owner ==. uid &&. query) ;
  titles <- project Title wishes;
  respond titles
}
```

Safe!

I. Motivation

Why secure web applications?

II. Demonstration

How to secure apps with STORM?

III. Implementation

How does STORM enforce security?

I. Motivation

Why secure web applications?

II. Demonstration

How to secure apps with STORM?

III. Implementation

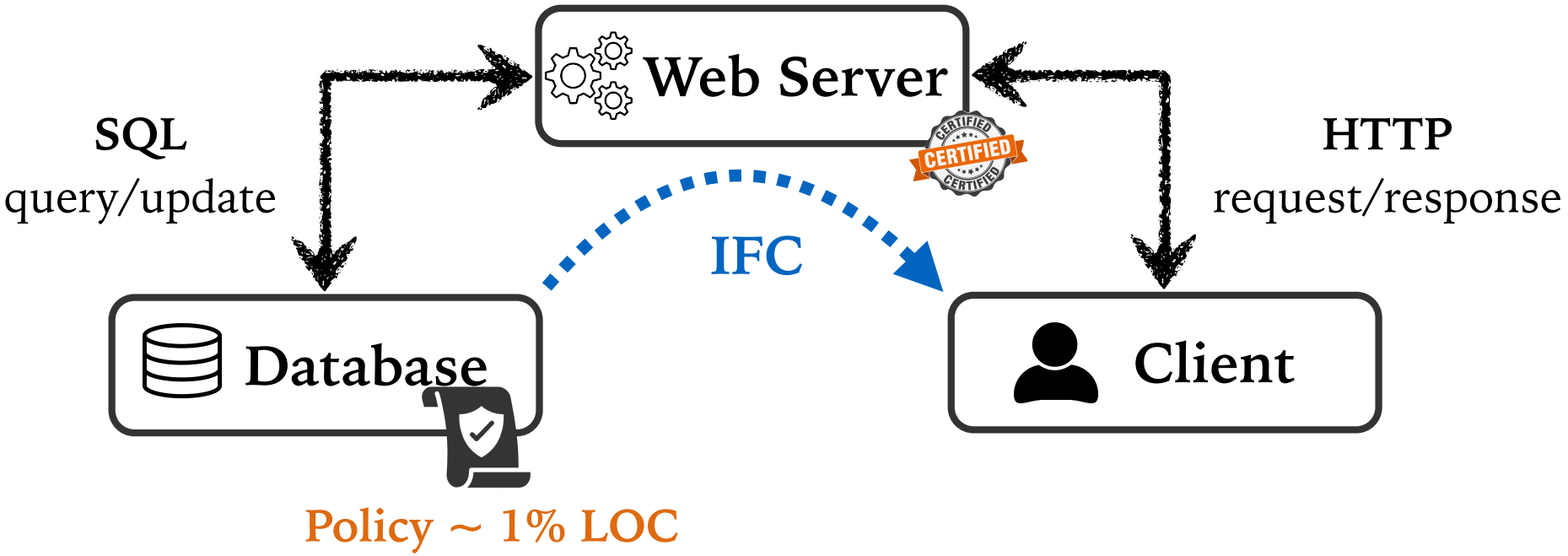
How does STORM enforce security?

IV. Evaluation

Can STORM secure real web-apps?

STORM

Security Typed ORM



Information Flow Control

By refinement type checking

Anatomy of a Server

```
showWishes uid = do {
  viewer <- getSessionUser();
  let qry = viewer == uid ? true : Level ==. “public”;
  wishes <- select (Owner ==. uid &&. qry) ;
  titles <- project Title wishes;
  respond titles
}
```

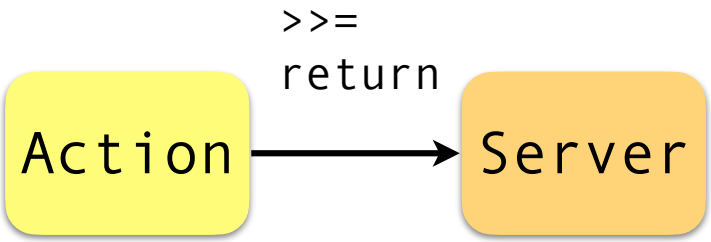
Anatomy of a Server

```
showWishes uid = do {
  viewer <- getSessionUser();
  let qry = viewer == uid ? true : Level ==. “public”;
  wishes <- select (Owner ==. uid &&. qry) ;
  titles <- project Title wishes;
  respond titles
}
```

Server

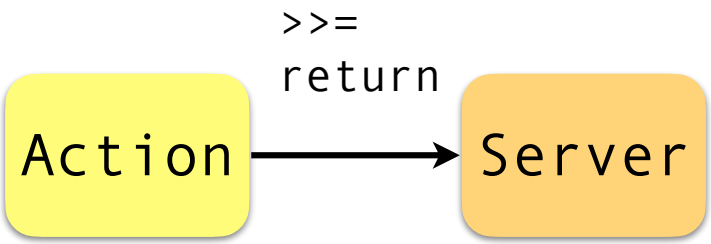
Anatomy of a Server

```
showWishes uid = do {
  viewer <- getSessionUser();
  let qry = viewer == uid ? true : Level ==. “public”;
  wishes <- select (Owner ==. uid &&. qry) ;
  titles <- project Title wishes;
  respond titles
}
```



Anatomy of a Server

```
showWishes uid = do {
  viewer <- getSessionUser();
  let qry = viewer == uid ? true : Level ==. “public”;
  wishes <- select (Owner ==. uid &&. qry)
  titles <- project title wishes;
  respond titles
}
```



Anatomy of a Server

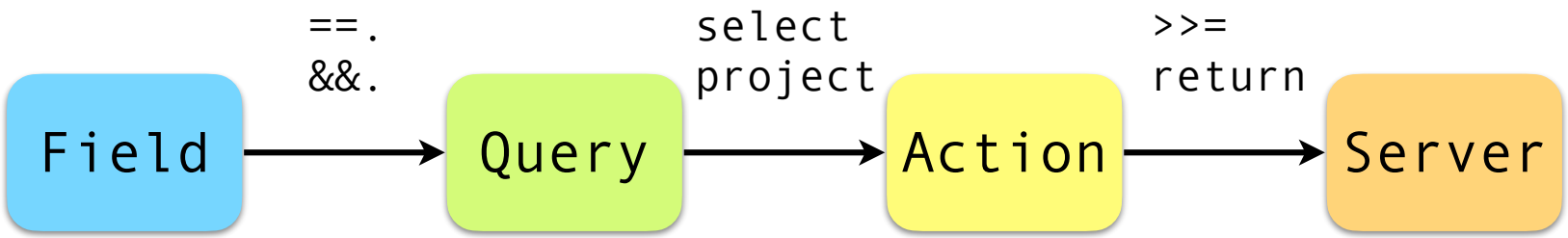
```
showWishes uid = do {
  viewer <- getSessionUser();
  let qry = viewer == uid ? true : Level ==. “public”;
  wishes <- select (Owner ==. uid &&. qry)
  titles <- project title wishes;
  respond titles
}
```

Anatomy of a Server

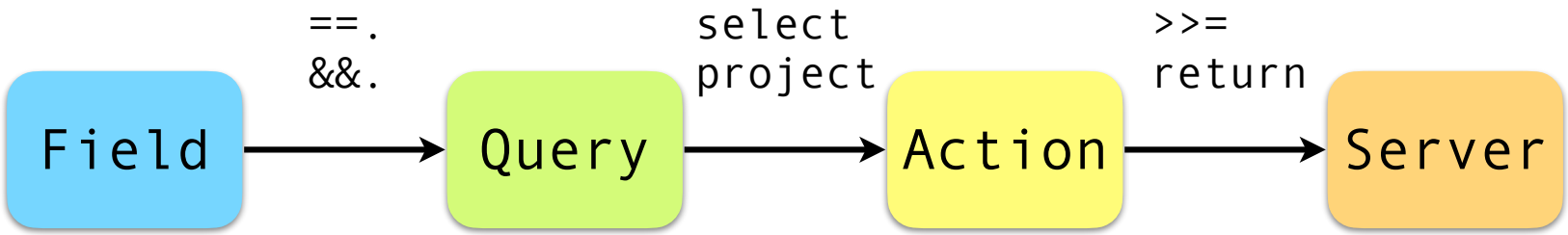
```
showWishes uid = do {
  viewer <- getSessionUser();
  let qry = viewer == uid ? true : Level ==. “public”;
  wishes <- select (Owner ==. uid &&. qry) ;
  titles <- project title wishes;
  respond titles
}
```

Anatomy of a Server

```
showWishes uid = do {
  viewer <- getSessionUser();
  let qry = viewer == uid ? true : Level ==. “public”;
  wishes <- select (Owner ==. uid &&. qry) ;
  titles <- project title wishes;
  respond titles
}
```



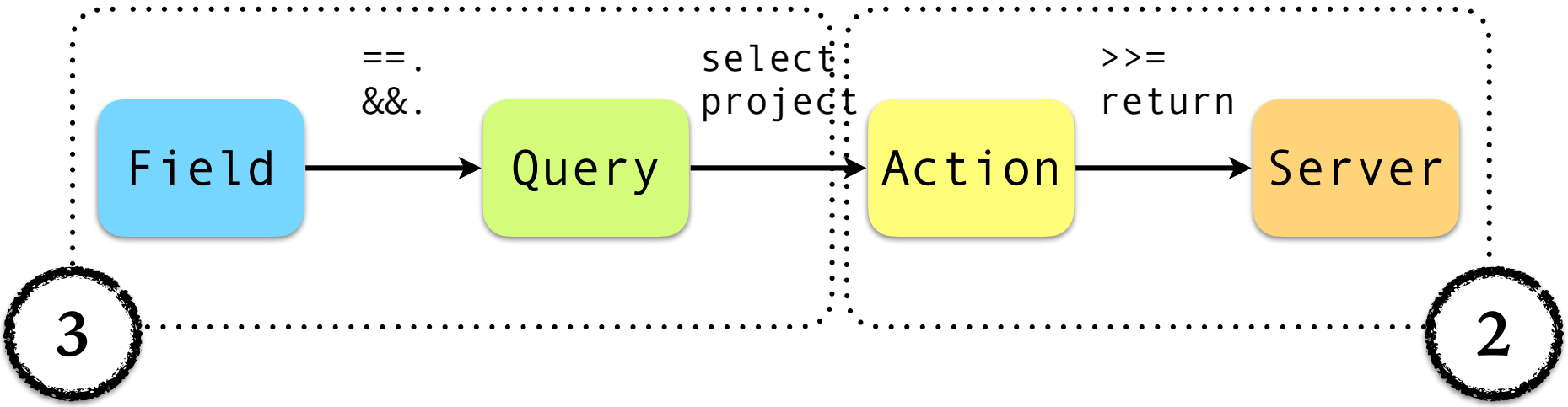
Anatomy of STORM



Refinement Typed API

Security by static type checking

Anatomy of STORM



Refinement Typed API

Security by static type checking

1

Anatomy of STORM

Refinement Typed API

Security by static type checking

1

Refinement Typed API

```
type Nat = {x:Int|0<=x}
```

`Int` values that are non-negative

Refinement Typed API

```
{x:Int|0<=x} → {v:Int|x<=v}
```

Function type is a contract

Refinement Typed API

"Pre-condition"

```
{x:Int|0<=x} → {v:Int|x<=v}
```

Function type is a contract

Refinement Typed API

"Post-condition"

```
{x:Int|0<=x} → {v:Int|x<=v}
```

Function type is a contract

Refinement Typed API

```
double  :: {x:Int | 0 <= x} → {v:Int | x <= v}
double x = 2 * x
```

Typing via SMT Validity Checking

$$\forall x, v\,.\; 0 \leq x \;\Rightarrow\; v = 2 \times x \;\Rightarrow\; x \leq v$$

Refinement Typed API

```
quad :: {x:Int | 0 <= x} → {v:Int | x <= v}

quad = double ∘ double
```

How to type "compose" ?

```
f ∘ g = \x → f (g x)
```

Refinement Typed API

```
f ∘ g = \x → f (g x)
```

Refinement Bounds [*Vazou et al. ICFP 15*]

```
(∘) :: (Cmp p q r) ⇒ (y:b → {v:c|q y v})
                   → (x:a → {v:b|p x v})
                   → (x:a → {v:c|r x v})
where
  Cmp p q r = \x y z → p x y ⇒ q y z ⇒ r x z
```

Refinement Bounds [*Vazou et al. ICFP 15*]

```
(∘) :: (Cmp p q r) ⇒ (y:b → {v:c|q y v})
                   → (x:a → {v:b|p x v})
                   → (x:a → {v:c|r x v})
where
  Cmp p q r = ∀x y z. p x y ⇒ q y z ⇒ r x z
```

Refinement Parameters

Related by Horn Constraint

Instantiated at use by Liquid Typing

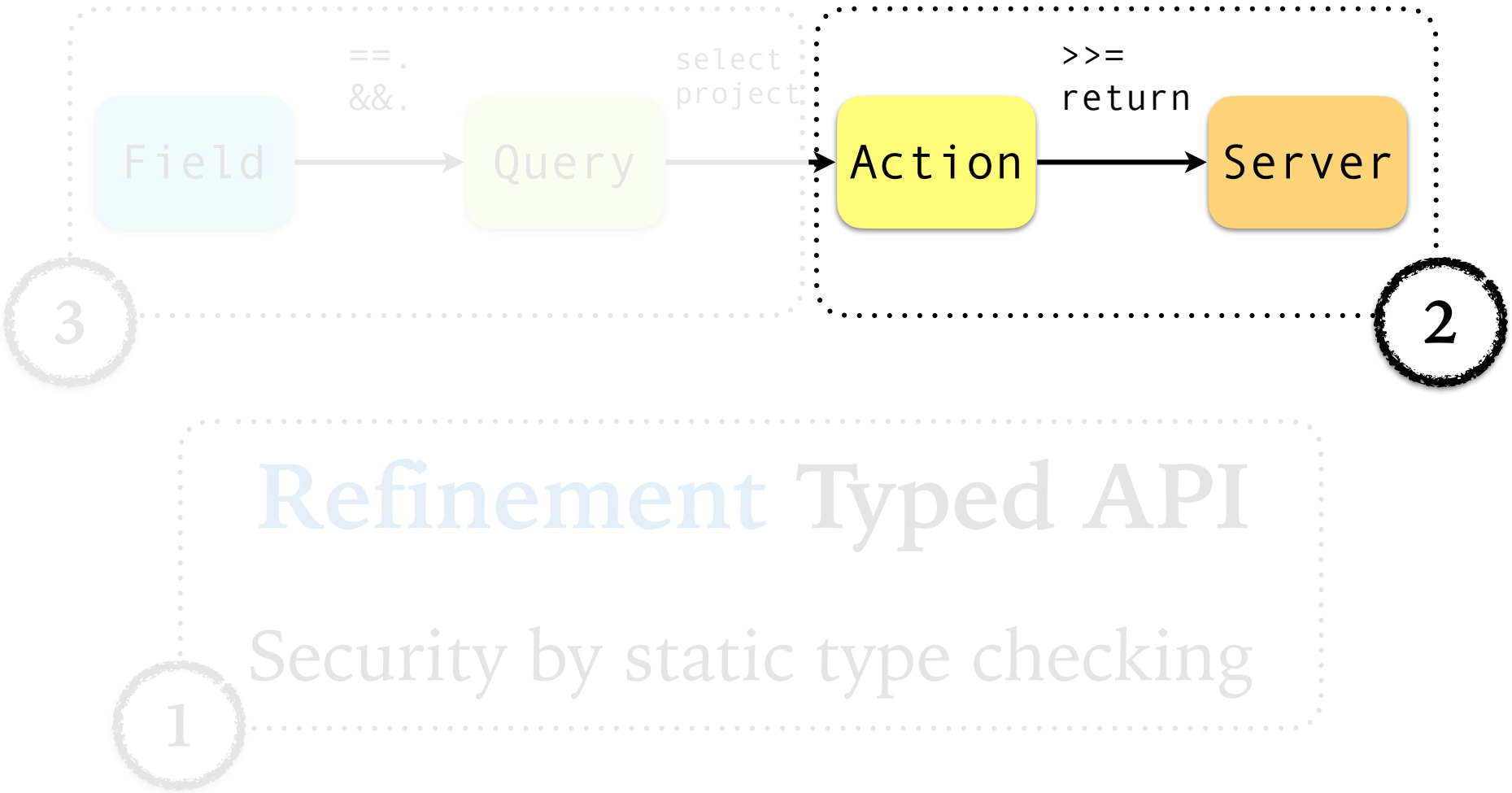
Anatomy of STORM

Refinement Typed API

Security by static type checking

1

Anatomy of STORM



Authorizees & Observers

[*Polikarpova et al. ICFP 20*]

```
showWishes uid = do {
  wishes <- select (Owner ==. uid) ;
  titles <- project Title wishes;
  respond titles
}
```

Auth

Authorizees

Set of users *authorized to access* data

Authorizees & Observers

[*Polikarpova et al. ICFP 20*]

```
showWishes uid = do {
  wishes <- select (Owner ==. uid) ;
  titles <- project Title wishes;
  respond titles
}
```

Auth

Authorizees

$$Auth \equiv \lambda u \rightarrow u = uid$$

Authorizees & Observers

[*Polikarpova et al. ICFP 20*]

```
showWishes uid = do {
  wishes <- select (Owner ==. uid) ;
  titles <- project Title wishes;
  respond titles
}
```

Obs

Observers

Set of users *provided access to* data

Authorizees & Observers

[*Polikarpova et al. ICFP 20*]

```
showWishes uid = do {
  wishes <- select (Owner ==. uid) ;
  titles <- project Title wishes;
  respond titles
}
```

Obs

Observers

$$Obs \equiv \lambda u \rightarrow u = sessionUser$$

Authorizees & Observers

[*Polikarpova et al. ICFP 20*]

```
showWishes uid = do {
  wishes <- select (Owner ==. uid) ;
  titles <- project Title wishes;
  respond titles
}
```

Auth

Obs

Policy Enforcement

$$Obs \subseteq Auth$$

Authorizees & Observers

[*Polikarpova et al. ICFP 20*]

```
showWishes uid = do {
  wishes <- select (Owner ==. uid) ;
  titles <- project Title wishes;
  respond titles
}
```

Auth

Obs

Policy Enforcement

$$\forall u \,.\, u = sessionUser \Rightarrow u = uid$$

Authorizees & Observers

[*Polikarpova et al. ICFP 20*]

```
showWishes uid = do {
  viewer <- getSessionUser();
  let qry = viewer == uid ? true : Level ==. “public”;
  wishes <- select (Owner ==. uid &&. qry) ;
  titles <- project Title wishes;
  respond titles
}
```

Auth

Obs

Policy Enforcement

$$\lambda u \rightarrow u = sessionUser \subseteq \lambda u \rightarrow True$$

Authorizees & Observers

[*Polikarpova et al. ICFP 20*]

```
showWishes uid = do {
  viewer <- getSessionUser();
  let qry = viewer == uid ? true : Level ==. "public";
  wishes <- select (Owner ==. uid &&. qry) ;
  titles <- project Title wishes;
  respond titles
}
```

Auth

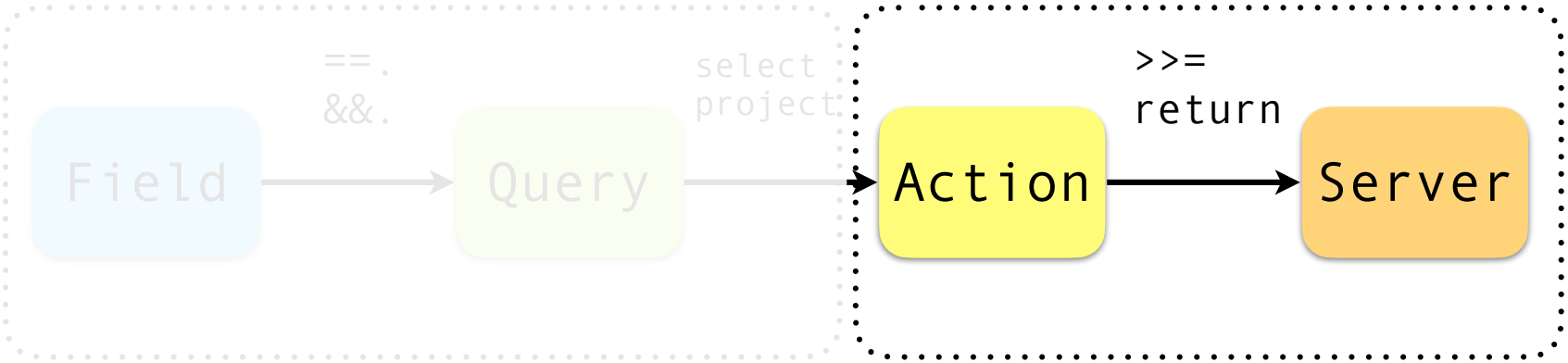
Obs

Policy Enforcement

$$\forall u \,.\, u = sessionUser \Rightarrow True$$

Authorizees & Observers

[*Polikarpova et al. ICFP 20*]



Effects in the `Action` monad

("Ghost" assertions for each statement)

Effects in the `Action` monad

```
showWishes uid = do {
  wishes <- select (Owner ==. uid);
  titles <- project Title wishes;
  respond titles
}
```

Monadic Computations Yielding `t` Values

`Action t`

Effects in the `Action` monad

```
showWishes uid = do {
  wishes <- select (Owner ==. uid);
  titles <- project Title wishes;
  respond titles
}
```

`Action` Refined by "Ghost" Security Effects

`Action <`*auth*`,` *obs*`> t`

`Action` Refined by "Ghost" Security Effects

STORM API: `return`

```
a → Action <λu → True , λu → False> a
```

Pure action doesn't access or send data

auth = All users *obs* = No Users

`Action` Refined by "Ghost" Security Effects

STORM API: `respond`

```
Text → Action <λu → True, λu → u = sessionUser> ()
```

Does not access sensitive data

auth = All users *obs* = Session User

`Action` Refined by "Ghost" Security Effects

STORM API: "Sequence" `(>>=)`

```
Action a → (a → Action b) → Action b
```

`Action` Refined by "Ghost" Security Effects

STORM API: "Sequence" `(>>=)`

$$\texttt{Action<}a_1\texttt{,}o_1\texttt{>\ a} \rightarrow \texttt{(a} \rightarrow \texttt{Action<}a_2\texttt{,}o_2\texttt{>\ b)} \rightarrow \texttt{Action<}a\texttt{,}o\texttt{>\ b}$$

`Action` Refined by "Ghost" Security Effects

STORM API: "Sequence" `(>>=)`

```
(Sub o2 a1, And a1 a2 a, Or o1 o2 o) ⇒

Action<a1,o1> a → (a → Action<a2,o2> b) → Action<a,o> b
```

`Action` Refined by "Ghost" Security Effects

STORM API: "Sequence" `(>>=)`

```
(Sub o2 a1, And a1 a2 a, Or o1 o2 o) ⇒
Action <a1, o1> a → (a → Action <a2, o2> b) → Action <a, o> b
```

Refinement Bound* `Sub` o_2 a_1

Require *policy enforced* at each sequencing

* see [*Vazou et al. ICFP 15*]

`Action` Refined by "Ghost" Security Effects

STORM API: "Sequence" `(>>=)`

```
(Sub o2 a1, And a1 a2 a, Or o1 o2 o) ⇒
Action<a1,o1> a → (a → Action<a2,o2> b) → Action<a,o> b
```

Refinement Bound* `And` a_1 a_2 a

Ensures output's authorizees are *intersection* of inputs'

* see [*Vazou et al. ICFP 15*]

`Action` Refined by "Ghost" Security Effects

STORM API: "Sequence" `(>>=)`

```
(Sub o2 a1, And a1 a2 a, Or o1 o2 o) ⇒
Action <a1,o1> a → (a → Action <a2,o2> b) → Action <a,o> b
```

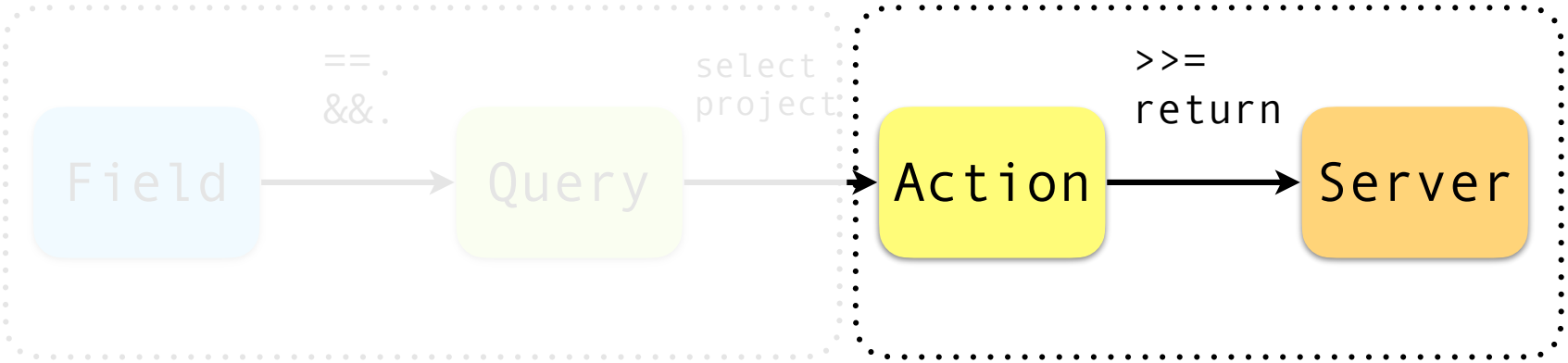
Refinement Bound* `Or` o_1 o_2 o

Ensures output's observers are *union* of inputs'

* see [*Vazou et al. ICFP 15*]

Authorizees & Observers

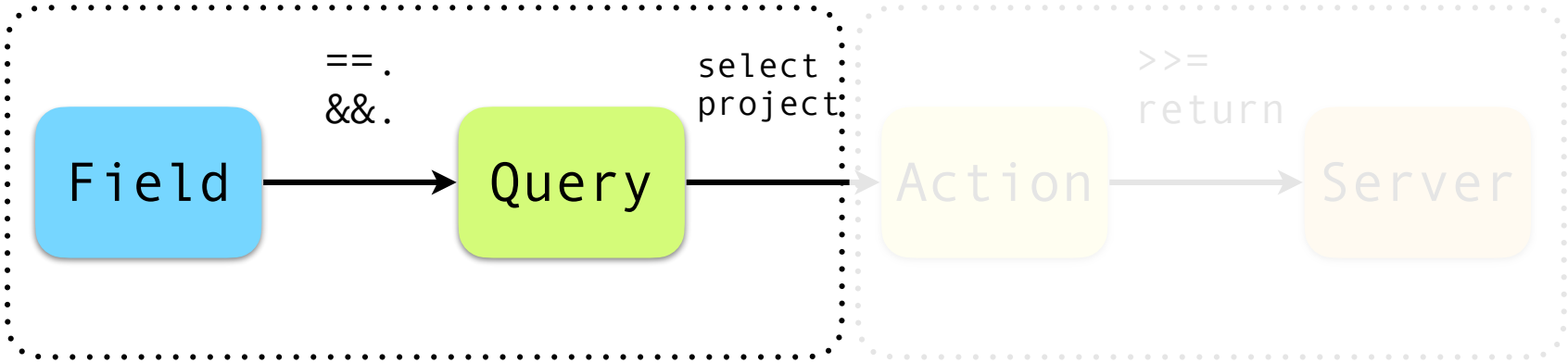
[*Polikarpova et al. ICFP 20*]



`Action` **Indexed with Security Effects**

How to track authorizees?

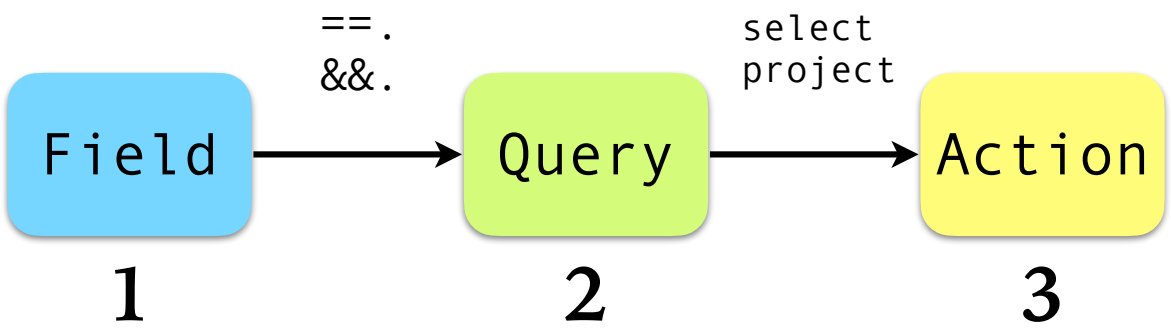
Anatomy of STORM



Refinement Typed API

How to track authorizees?

How to track Authorizees



1. **Inject:** Policies in database `Field`
2. **Propagate:** When building `Query`
3. **Extract:** *Auth* at `Query` execution

1. **Inject:** Policies in database `Field`

DB Schema

```
User
  name  String
  email String

Wish
  owner UserId
  title String
  level String
```

ORM Fields [Persistent]

```
Name  :: Field User String
Email :: Field User String


Owner :: Field Wish UserId
Title :: Field Wish String
Level :: Field Wish String
```

Column in table `row` with data of type `val`

```
Field row val
```

1. **Inject:** Policies in database `Field`

Refine `Field` with *authorizees* & *representation

```
Field <pol, rep> row val
```

Refinements are `row` dependent

```
pol : row → user → Bool
rep : row → val  → Bool
```

* Represent SQL query semantics at refinement type level

1. **Inject:** Policies in database `Field`

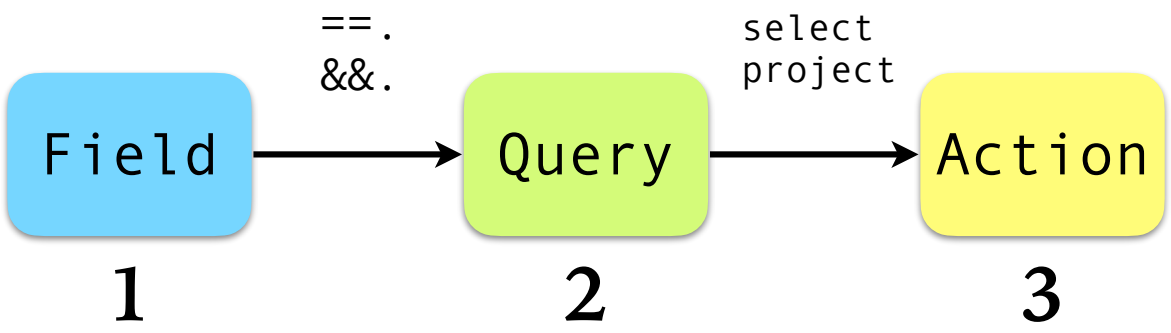
Refine `Field` **with** ***authorizees*** **&** ***representation****

`Field <`pol`,` rep`> row val`

```
Owner :: Field <All,  λr v → v = owner r> Wish UserId
Title :: Field <Own, λr v → v = title r>  Wish String
Level :: Field <All,  λr v → v = level r> Wish String
```

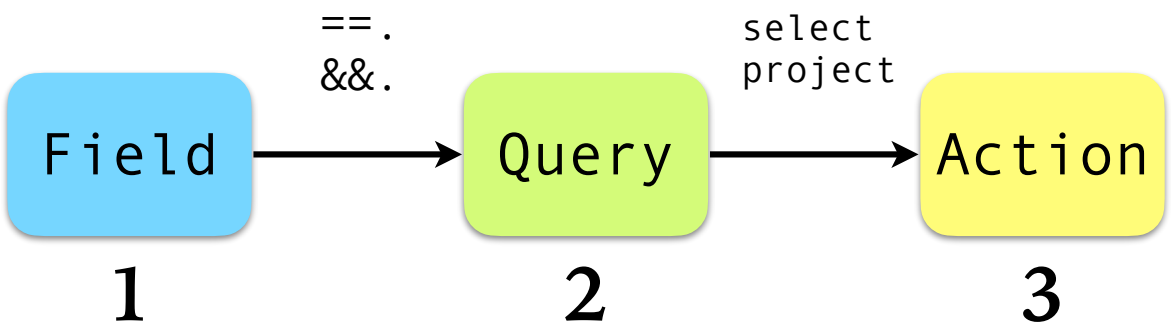
$All \equiv \lambda r\ u \to True$ $\quad$ $Own \equiv \lambda r\ u \to owner\ r = u \vee level\ r = \text{public}$

How to track Authorizees



1. **Inject:** Policies in database `Field`
2. **Propagate:** When building `Query`
3. **Extract:** *Auth* at `Query` execution

How to track Authorizees



1. **Inject:** Policies in database `Field`

2. **Propagate:** When building `Query`

3. **Extract:** *Auth* at `Query` execution

2. **Propagate:** When building `Query`

A `Query` to find “public” `Wishes` of `uid`

```
qry :: Query Wish
qry = Owner ==. uid &&. Level ==. “public”
```

2. **Propagate:** When building `Query`

Refine `Query` **by** ***authorizees*** **&** ***result invariant****

`Query <`*pol*`,` *inv*`> row`

* Represent SQL query semantics at refinement type level

2. **Propagate:** When building `Query`

Refine `Query` by *authorizees* & *result invariant

`Query` <*Own*, $\lambda r \to \mathit{level}\ r = \text{public}$> `Wish`

Authorizees: Owner unless level is public

Invariant: Every result row's level is public

$$\mathit{Own} \equiv \lambda r\ u \to \mathit{owner}\ r = u \vee \mathit{level}\ v = \text{public}$$

2. **Propagate:** When building `Query`

Refine `Query` by *authorizees* & *result invariant*

`Query <`*pol*`,` *inv*`> row`

Ensured by `Query` builder API

`(==.)`, `(<.)`, `(&&.)`, `(||.)`, *etc*.

2. **Propagate:** When building `Query`

Refine `Query` by *authorizees* & *result invariant*

`Query` builder API `(==.)`

```
(FldEq rep inv) ⇒
Field <pol,rep> row val → val → Query <pol,inv> row
```

$$\texttt{FldEq}\ rep\ inv = \forall r, v\,.\,(rep\ r\ v) \Rightarrow (inv\ r)$$

2. **Propagate:** When building `Query`

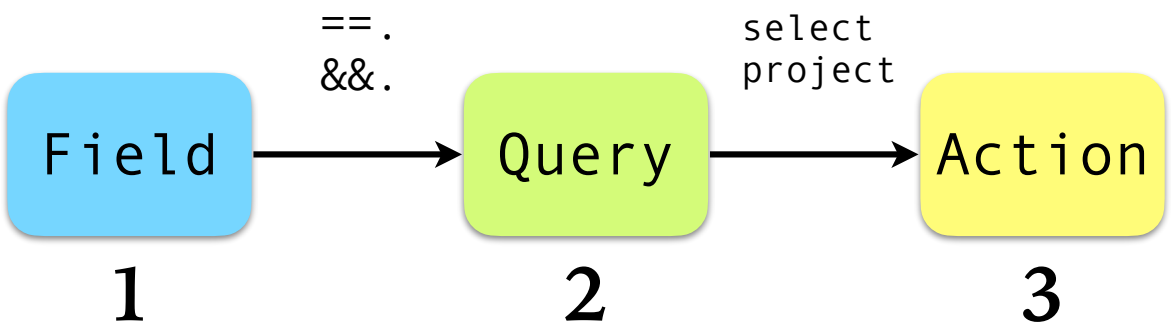
Refine `Query` by *authorizees* & *result invariant*

`Query` builder API `(&&.)`

```
(And pol1 pol2 pol, And inv1 inv2 inv) ⇒
Query <pol1,inv1> row → Query <pol2,inv2> row → Query <pol,inv> row
```

Output *pol* and *inv* are conjunction of inputs'

How to track Authorizees

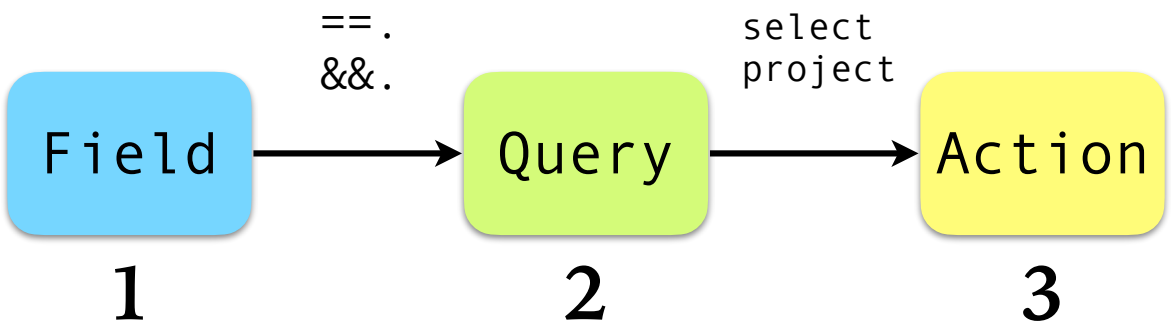


1. **Inject:** Policies in database `Field`

2. **Propagate:** When building `Query`

3. **Extract:** *Auth* at `Query` execution

How to track Authorizees

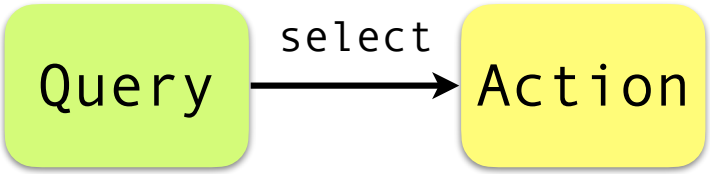


1. **Inject:** Policies in database `Field`

2. **Propagate:** When building `Query`

3. **Extract:** *Auth* at `Query` execution

3. Extract: *Auth* at `Query` execution



`Query` **execution API:** `select`

3. Extract: *Auth* at `Query` execution

`Query` execution API: `select`

```
Query row → Action [row]
```

3. Extract: *Auth* at `Query` execution

`Query` execution API: `select`

```
Query <pol, inv> row → Action <auth, None> [row]
```

3. Extract: *Auth* at `Query` execution

`Query` execution API: `select`

```
(ViewPolicy inv auth pol) ⇒
Query <pol, inv> row → Action <auth, None> [row]
```

Bound `ViewPolicy`: *Invariant rows' *authorizees* satisfy *policy***

$$\texttt{ViewPolicy}\ inv\ auth\ pol = \forall r, u\,.\,(inv\ r) \Rightarrow (auth\ u) \Rightarrow (pol\ r\ u)$$

* Not *all* rows, only the *inv* rows returned by SQL query!

3. Extract: *Auth* at `Query` execution

`Query` execution API: `select`

IF `qry :: Query <`Own`,` $\lambda r \rightarrow level\ r = \text{public}$`> Wish`

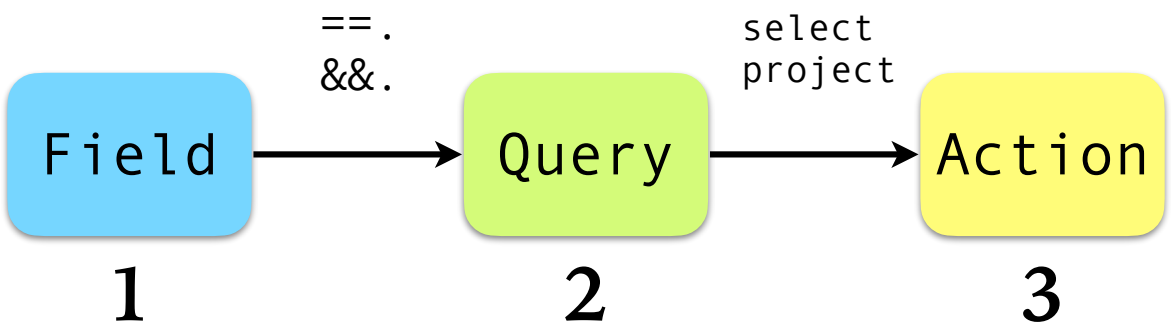
THEN `select qry :: Action <`$\lambda u \rightarrow True$`,` $None$`> [Wish]`

Authorizees of rows satisfying *invariant** satisfy *policy*

As $\forall r, u\,.\,(level\ r = \text{public}) \Rightarrow True \Rightarrow (owner\ r = u \vee level\ r = \text{public})$

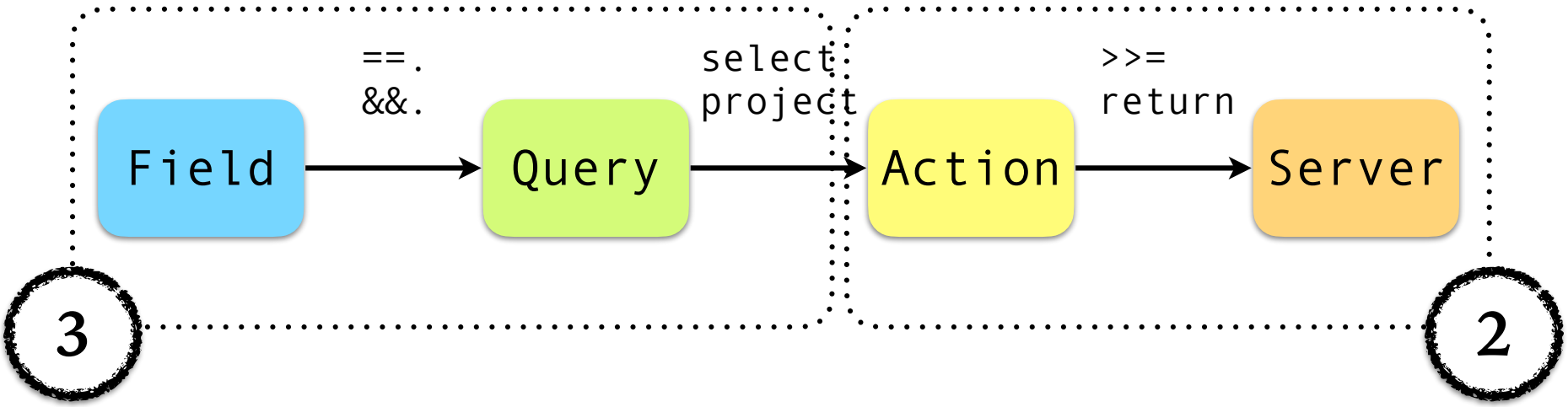
* Not *all* rows, only the *inv* rows returned by SQL query!

How to track Authorizees



1. **Inject:** Policies in database `Field`
2. **Propagate:** When building `Query`
3. **Extract:** *Auth* at `Query` execution

Anatomy of STORM



Refinement Typed API

Tracking *authorizees* & *observers*

1

I. Motivation

Why secure web applications?

II. Demonstration

How to secure apps with STORM?

III. Implementation

How does STORM enforce security?

IV. Evaluation

Can we secure web-apps with STORM?

I. Motivation

Why secure web applications?

II. Demonstration

How to secure apps with STORM?

III. Implementation

How does STORM enforce security?

IV. Evaluation

Can STORM secure real web-apps?

IV. Evaluation

Expressive

Does STORM capture *interesting* policies?

Convenient

How much *extra work* is needed to use STORM?

Auditable

Does STORM *reduce* the code we need to get right?

Expressive

Does STORM capture *interesting* policies?

System	Benchmark	Model	Policy
UrFlow	`secret`	8	9
	`poll`	14	16
	`calendar`	15	29
	`gradebook`	18	24
	`forum`	19	34
Jacqueline	`conference*`	42	46
	`course`	32	11
	`health`	79	23
Hails	`gitsar`	16	21
LWeb	`bibifi`	312	101
Total		555	314

Ported policies from 10 benchmarks from 4 previous frameworks

Expressive

Does STORM capture *interesting* policies?

System	Benchmark	Model	Policy
UrFlow	`secret`	8	9
	`poll`	14	16
	`calendar`	15	29
	`gradebook`	18	24
	`forum`	19	34
Jacqueline	`conference*`	42	46
	`course`	32	11
	`health`	79	23
Hails	`gitsar`	16	21
LWeb	`bibifi`	312	101
Total		555	314

Static

But not IFC

Expressive

Does STORM capture *interesting* policies?

System	Benchmark	Model	Policy
UrFlow	secret	8	9
	poll	14	16
	calendar	15	29
	gradebook	18	24
	forum	19	34
Jacqueline	conference*	42	46
	course	32	11
	health	79	23
Hails	gitsar	16	21
LWeb	bibifi	312	101
Total		555	314

Static

But not IFC

Dynamic

Overhead & late checks

* STORM supports all but one policy that relies on *missing* DB rows

IV. Evaluation

Expressive

Does STORM capture *interesting* policies?

Convenient

How much *extra work* is needed to use STORM?

Auditable

Does STORM *reduce* the code we need to get right?

Convenient

How much *extra work* is needed to use STORM?

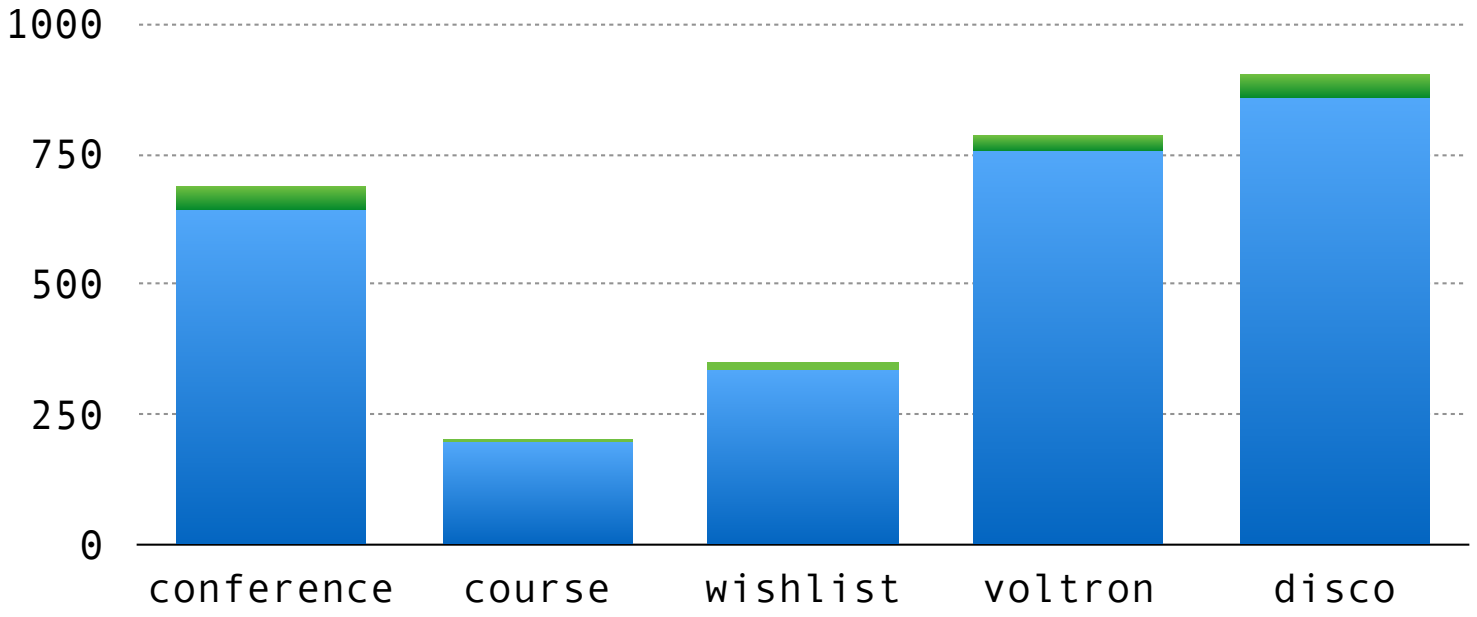
```
getAuthors :: p:Paper -> Action <\u -> PcOrAuthOrAccepted p u, None> [Text]

getAuthors paper = do {

  (paperId, authorId) <- project2 (PaperId, PaperAuthor) paper;

  author              <- select (UserId ==. authorId);
  authors             <- mapT (project UserName) author;

  coauthors           <- select (PaperCoauthorPaper ==. paperId);
  coauthorNames       <- mapT (project PaperCoauthorAuthor) coauthors;

  return $ authors ++ coauthorNames;
}
```

Type Annotations

Specify server function's *authorizees* & *observers*

Convenient

How much *extra work* is needed to use STORM?



Type Annotations

1 line of **annotation** per **20** lines of **code**

IV. Evaluation

Expressive

Does STORM capture *interesting* policies?

Convenient

How much *extra work* is needed to use STORM?

Auditable

Does STORM *reduce* the code we need to get right?

Auditable

Does STORM *reduce* the code we need to get right?

Disco & Voltron

We built two full-fledged JS web apps with UIs …

Auditable

Does STORM *reduce* the code we need to get right?

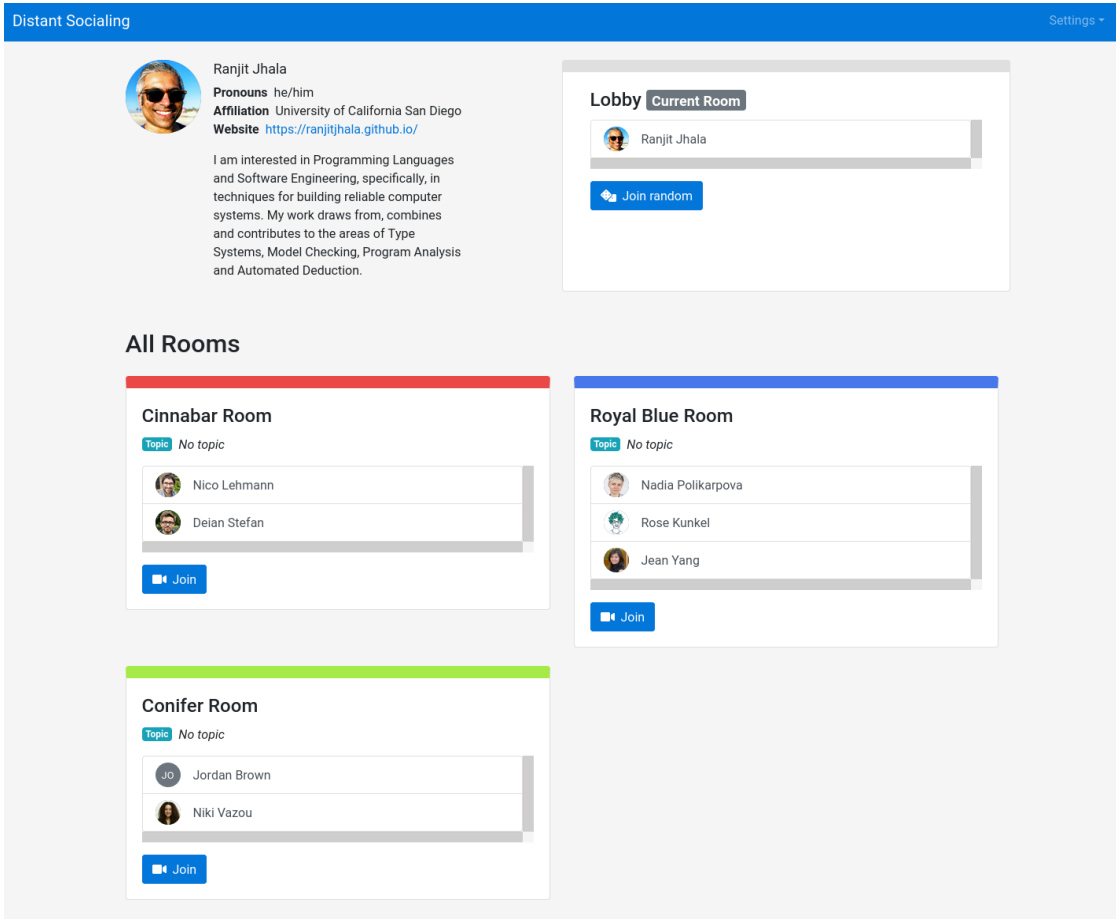
Server

Client

Disco & Voltron

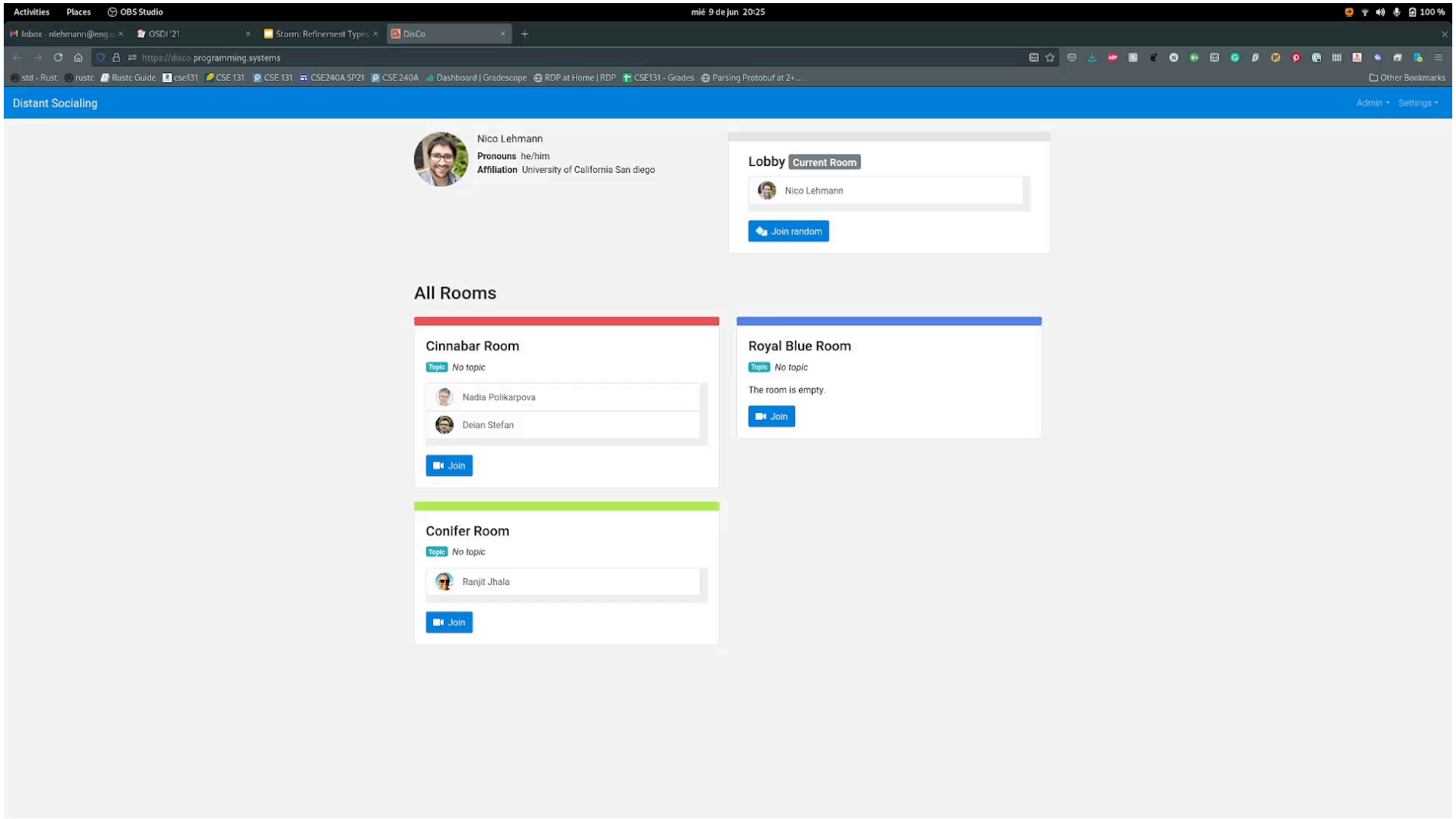
We built two full-fledged JS web apps with UIs …

Disco ("Virtual Hallway Track")

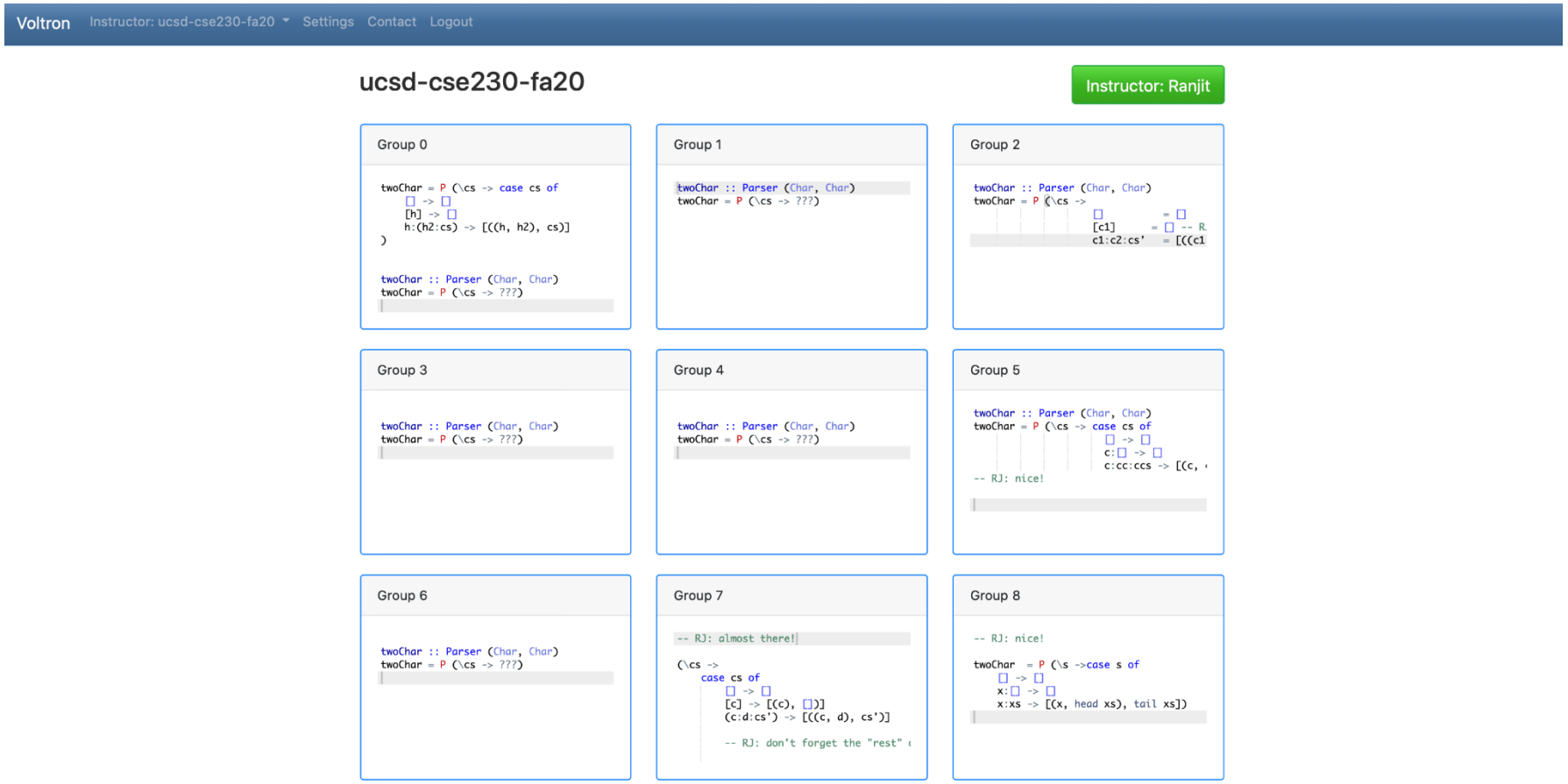


PLDI/PLMW(Jun 20) & CAV/VMW(Jul 20), ~100 Users

Disco ("Virtual Hallway Track")

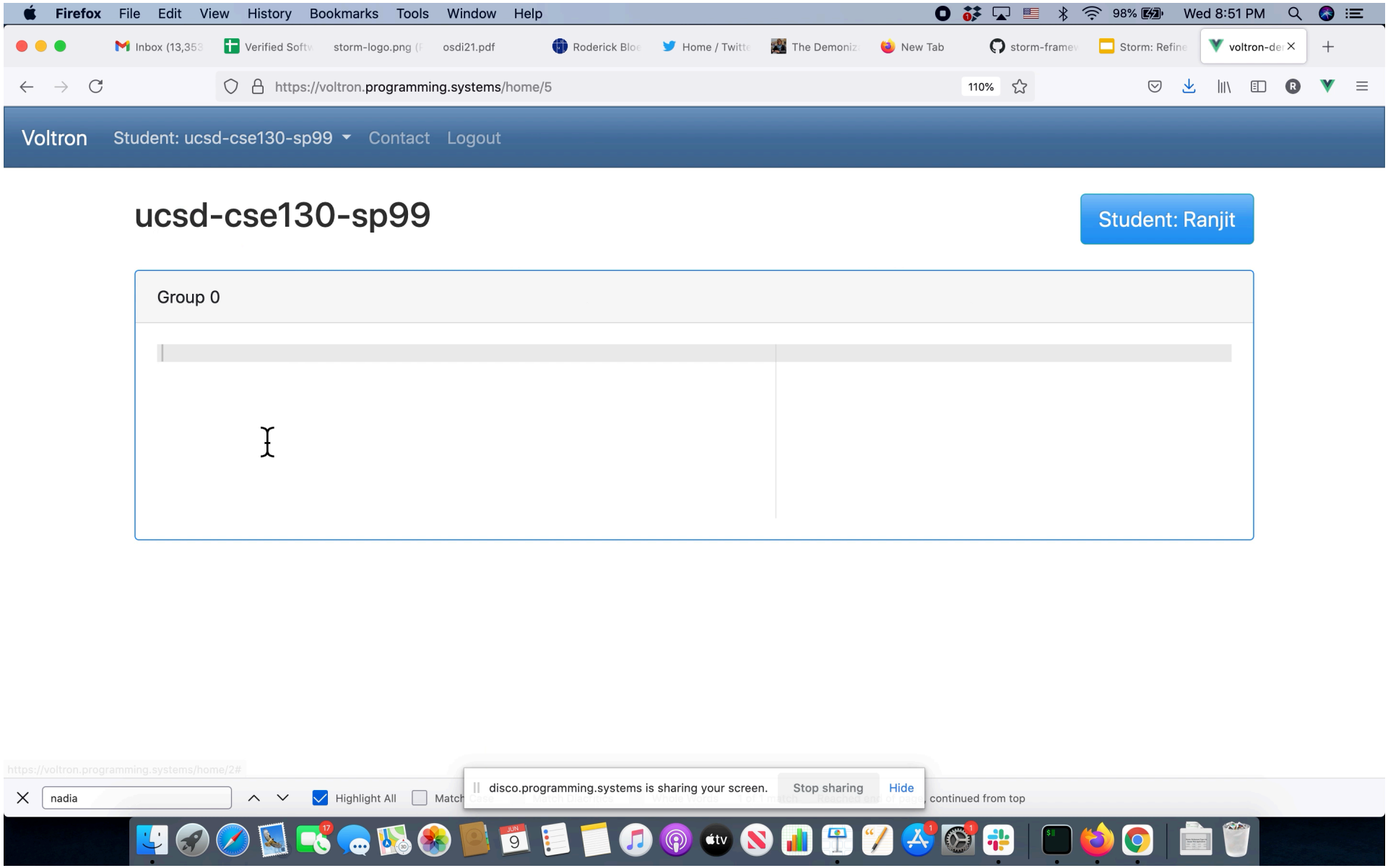


Voltron ("Real-time Group Editor")



Two instructors, 5 classes Fall 2020-21, 50 - 200 Students

Voltron ("Real-time Group Editor")



Auditable

Does STORM *reduce* the code we need to get right?

Application	LOC			
	Server	*Models*	*Policy*	*Client*
`voltron`	756	32	37	1012
`disco`	859	42	32	4630
Total	1615	74	69	5642

STORM Centralizes & Reduces Trusted Code*

Policy < 4% of *Server* (< 1% of *Server*+*Client*)

I. Motivation

Why secure web applications?

II. Demonstration

How to secure apps with STORM?

III. Implementation

How does STORM enforce security?

IV. Evaluation

Can STORM secure real web-apps?

Refinement Types for Secure Web Applications

https://storm-framework.github.io

STORM

Refinement Types for Secure Web Apps

https://storm-framework.github.io

GHC Plugin verifier runs at each compilation

For Details, Proofs, Case studies, Evaluation...

"Refinement Types for Secure Web Apps", *Lehmann et al. OSDI 2021*