STORM: Refinement Types for Secure Web Applications

Nico Lehmann
UC San Diego

Rose Kunkel
UC San Diego

Jordan Brown
Independent

Jean Yang
Akita Software

Niki Vazou
IMDEA Software Institute

Nadia Polikarpova
UC San Diego

Deian Stefan
UC San Diego

Ranjit Jhala
UC San Diego

Abstract

We present STORM, a web framework that allows developers to build MVC applications with compile-time enforcement of centrally specified data-dependent security policies. STORM ensures security using a *Security Typed ORM* that refines the (type) abstractions of each layer of the MVC API with logical assertions that describe the data produced and consumed by the underlying operation and the users allowed access to that data. To evaluate the security guarantees of STORM, we build a formally verified reference implementation using the Labeled IO (LIO) IFC framework. We present case studies and end-to-end applications that show how STORM lets developers specify diverse policies while centralizing the trusted code to under 1% of the application, and statically enforces security with modest type annotation overhead, and no run-time cost.

1 Introduction

We trust web applications with our most sensitive data: our finances, health records, email, or even our participation in political protests. While application developers go to great lengths to protect this data, today's approach to safeguarding sensitive data by sprinkling access control checks throughout the application is not working. Even companies with dedicated security teams are failing. For example, in 2018 Facebook accidentally allowed third-party applications to access the photos of 6.8 million users without their explicit permission [1]. This was not their first (nor last) leak. And Facebook is not unique: *sensitive data exposure* and *broken access control* are—and have been for almost a decade—on the OWASP top ten list of most common web application vulnerabilities [2, 3].

To fundamentally address this class of bugs, we need to reduce the amount of code developers need to get right. One promising approach to doing this is to *centralize policy specification*, i.e., specify data access control policies in a centralized place, and enforce policies *automatically*. This could reduce the code developers need to get right from the whole application—as a single missing check could introduce a vulnerability—to the policy specification code.

Centralizing policy specification is not a new idea. Several web frameworks (e.g., HAILS [4], JACQUELINE [5], and LWEB [6]) already do this. These frameworks, however, have two shortcomings that have hindered their adoption. First, they enforce policies *at run-time*, typically using dynamic information flow control (IFC). While dynamic enforcement is better than no enforcement, dynamic IFC imposes a high performance overhead, since the system must be modified to track the provenance of data and restrict where it is allowed to flow. More importantly, certain policy violations are only discovered once the system is deployed, at which point they may be difficult or expensive to fix, e.g., on applications running on IoT devices [7].

Second, these frameworks are *invasive*—they typically require modifications to the language runtime and database object-relational mapping (ORM). For example, JACQUELINE uses a *faceted* ORM and runtime to keep track of multiple facets of any individual value and only shows the right facet to the right user (e.g., when reading a password, a user can see their own password but get a default facet when trying to read another user's password). HAILS and LWEB, on the other hand, use *labeled values* at the ORM and language level to restrict the flow of sensitive, labeled data. This means that developers need to write code that is aware of faceted or labeled values, i.e., they need to write code that is aware of the underlying IFC enforcement mechanism. Worse, this invades policy specification. For example, in HAILS, developers can't simply write declarative policies, they often need to use the low-level APIs used to track and enforce IFC to, for example, inspect and manipulate labeled values [4, 8]. This not only increases the amount of *code* they need to get right, but also makes it hard to get the *policy* right since manipulating labeled values is still an IFC expert—and not web developer—task.

We built the STORM web framework to address these shortcomings. With STORM, users specify all security policies in a declarative language, alongside the *data model*, the description of the application database schema. Policies are *logical assertions* that describe which users are allowed to view, insert, or update particular rows and columns of each table in the database. STORM enforces these policies *statically*, at compile-time—and *non-invasively*, without translating them to labels or facets. This means that (1) STORM does not impose any run-time overhead, (2) developers can catch bugs due to policy violations (e.g., where the application incorrectly handles sensitive data) early, and (3) they don't need to understand the details of the underlying enforcement mechanism to specify or audit policy code.

STORM statically enforces policies using *refinement types* [9]: types decorated with logical assertions that can constrain values, e.g., to say that an `Int` is non-negative or that a `User` is the author of a `Paper`. Our key insight is to refine STORM's API with logical assertions that describe the data produced and consumed by the underlying operation and the users allowed access to that data. We use this insight to realize STORM via four contributions.

1. Design (§ 3) Our first contribution is a novel design that enriches the data model with a declarative policy—the *refined data model*—to generate an application-specific ORM layer, which STORM annotates with refinement types that reflect the security policies. To our knowledge, this is the first framework to statically and unobtrusively enforce policies previously thought to only be expressible using runtime enforcement.

2. Implementation (§ 5) Our second contribution is an implementation of STORM in Haskell that uses LIQUIDHASKELL, an off-the-shelf refinement type checker to *statically and automatically verify* whether the application code using the security-typed ORM—e.g., code handling user requests and rendering HTML responses—adheres to the policies. STORM does this without imposing any invasive changes to the language runtime or database ORM interface. At most, developers write (untrusted and verified) light-weight type annotations to help the checker prove their code does not leak.

3. Verification (§ 6) Our third contribution is a formally verified reference implementation that proves that the STORM API is secure by showing how to reduce a well-typed STORM program into an LIO [10] program that never throws security exceptions. This allows us to carry over the previously mechanized non-interference results from LIO [6, 10] to show that well-typed programs cannot leak or corrupt sensitive data.

4. Evaluation (§ 7) Our final contribution is an empirical evaluation of the *expressiveness* of STORM's policy mechanism, the programmer *effort* needed for static enforcement, and, ultimately, of the *reduction* in the amount of code the developer has to get right to not leak data in real web applications. First, we show that our centralized policy specification approach is expressive enough to describe, often more naturally, a large suite of policies from the literature. Second, we use STORM to write statically verified implementations of several case studies from the literature, including those that had previously only been amenable to dynamic policy enforcement, and show that the effort is modest: the programmer need only write 1 line of refinement type signatures per 20–30 lines of code (LOC). Third, we use STORM to build and deploy two new end-to-end web applications for collaborative text editing and video-based social interaction, that have been used at our university and at several academic workshops, respectively. We demonstrate that STORM distills the code that the developer has to get right to compact, auditable policies (under 70 LOC) that comprise under 1% of the application.

System	Audit	Static	Uninvasive	IFC
SWIFT [11]	✗	✓	✗	✓
SELINKS [12]	✓*	✓	✗	✗
RESIN [13]	✗	✗	✓*	✗
URFLOW [14]	✓	✓	✓	✓*
IFDB [15]	✓	✗	✗	✓
HAILS [4]	✓*	✗	✗	✓
JACQLN [5]	✓	✗	✗	✓
LWEB [6]	✓*	✗	✗	✓
DAISY [16]	✓	✗	✗	✓
STORM	✓	✓	✓	✓

Figure 1: We compare STORM to previous web frameworks along various design goals. **Audit**: are the policies centralized and easily auditable; **Static**: is the enforcement at compile-time; **Uninvasive**: does enforcement require changes to the run-time; and **IFC**: does the framework enforce information flow control. We write ✓*for almost-met goals.

2 Goals & Related Work

We designed STORM with several goals in mind. First, the framework should provide *information flow control* (IFC) security to prevent not only explicitly bad data flows, but also implicit leaks where publicly viewable results are conditioned on sensitive data. Second, the framework should enable a centralized, and hence, easily *auditable* policy specification. Third, to find errors early, provide design-time feedback and avoid run-time overhead, the framework should permit *static* enforcement via *automatic*, compile-time verification. Fourth, the framework should not require *invasive* modifications to language run-times, database ORMs, or libraries. STORM builds on previous work, summarized in Figure 1, which have made great strides towards these goals.

IFC There are many flavors of IFC with different trade-offs [17, 18]. Systems differ in *when* they enforce IFC: at run-time via labels [10, 19–22], faceted values [19, 22, 23], secure multi-execution [24], or at compile-time via types [12, 25–28], or static analysis [29], or a hybrid combination [30–32]. Even within the same category, these systems differ in granularity of enforcement—from fine-grained to coarse-grained [33, 34], and the kinds of policies users can specify [35]. The SWIFT [11] system uses a static IFC type system [30] to enforce compile time security, but does not integrate with database ORMs, and hence, lacks centralized auditable specifications. IFDB [15] and DAISY [16] show how to perform fine-grained IFC within DB systems, but are not static, and focus on databases—and are thus not complete frameworks for building applications. STORM draws inspiration from the HAILS [4], LWEB [6], and JACQUELINE [5] frameworks which enforce auditable IFC policies that are associated with the application's data model. However, these approaches all perform dynamic enforcement and require invasive changes to the DB layer or run-time.

Static Several static frameworks express data-dependent policies using dependent types [36–38, 38, 39], labels [11, 12, 28, 40], or first-order logic formulas [41]. All the above require the programmer to sprinkle policy specifications across the application controller and view code, which is error prone and makes auditing difficult. SELINKS [12] centralizes policies within special functions that un/wrap data with labels, but requires invasive changes to the DB and run-time to propagate labels and does not prevent implicit leaks. URFLOW [14] enables verification of centralized and auditable specifications without requiring invasive changes, by using a bespoke symbolic execution algorithm to statically verify that the generated SQL queries are (semantically) contained in some allowed set. However, to statically compute the SQL queries, URFLOW requires programmers to write their applications in a domain-specific language (DSL). Further, URFLOW's approach is insufficient for full IFC as it misses implicit flows through SQL queries (as illustrated in § 3.4). In contrast, STORM enforces full IFC via off-the-shelf refinement type checking for a general purpose language with a rich ecosystem with tools and libraries for networking, databases, data serialization, etc. STORM uses a statically typed API for monadic IFC in the style of [42, 43], specifically, the approach of LIFTY [44], a core calculus that shows how to track IFC with logical refinement types. Unlike STORM, LIFTY cannot be used to build secure applications: it does not have database APIs, a language to specify centralized policies, formal guarantees for data-dependent policies, or even a way to write executable code.

System-based Security Several frameworks employ privilege separation to run application components with least privilege [45–49]. Others like RESIN [13] and QAPLA [50] restrict access to data by modifying the run-time to use fine-grained discretionary access control, or use cryptography to provide data confidentiality, authenticity, and integrity in the presence of compromised application components [51–53], or use proxies to implement web application firewalls [54, 55]. While some of these approaches, e.g., the use of cryptography are complementary to our approach, without IFC, they cannot prevent leaks that STORM eliminates by construction.

3 Design

We illustrate the design of STORM with a `WishList` application where users can share wishes with followers. STORM uses the *model-view-controller* (MVC) paradigm, where an application has three key elements: *models* which describe the persistent data important to the application, typically stored in a database (DB) and accessed via an *Object-relational mapping* (ORM); *views* which describe how the data corresponding to, e.g., users' requests are rendered on webpages via some combination of CSS, HTML and JavaScript; and *controllers* that respond to user's requests by suitably querying the DB via the models API, to produce an HTML or JSON results.

3.1 Auditable Policies via Refined Models

The key innovation in STORM is to centralize data-dependent security policies with the data model, in a *refined models* file.

Models & Policies Figure 2 shows the refined models file for the `WishList` app. The left column describes the data schema, as a collection of three tables `User`, `Wish` and `Follower`. Each row of the `User` table comprises the user's `name` and `email` address. Each row of the `Wish` table has an `owner` that identifies the `User` that the wish belongs to, a text `description` of the wish, and a numeric `price`. Each row of the `Follower` table describes a tuple where `user1` *follows* `user2`, with the `status` column indicating whether a follow-request has been initiated (`"pending"`), accepted (`"ok"`) or rejected (`"no"`). STORM lets the programmer specify policies that govern which DB rows can be inserted and which DB columns can be read or updated. A *policy* is a predicate over a `row` and `user` that is `True` if the `user` has access and `False` otherwise. The policy predicate can refer to all the columns of the row (whose column the policy is attached to) and so the values of those other columns can be used to determine whether the user has access. For example, we specify that `Wish`es can only be inserted by their owners via the policy `@IsOwner` which holds when the `user` equals the `owner` of the row. Similarly, we specify that each `Wish`'s `description` and `price` should only be read by the `owner` unless they are explicitly public via the policy `@Public` which holds when the `user` is the owner or the `level` is `"public"`. (Ignore the shaded `Follow` for now: we will return to it in § 3.3.) Finally, we specify that only the owner is allowed to update the `description` and `price`.

Default Policies The programmer can associate **`default`** policies with all the rows and columns not explicitly constrained otherwise. For example, `Allow` grants access to all users, while `Deny` grants access to none. Hence, **`default`** `read @Allow` and **`default`** `insert, update @Deny` say that (unless otherwise specified) anyone can `read` every column, and no one can `insert` rows or `update` columns.

3.2 Access Control

Let's see how STORM enforces the `Public` policy. Figure 3 shows a controller `showWishes` that responds to a request to display the wish list for a given `user`. (For now, *ignore* the shaded code.) The controller uses the models API to create a `Query` of the form `Owner ==. user`, which it executes using the ORM

```
User
  name    Text
  email   Text

Wish
  owner   UserId
  descr   Text
  level   Text
  price   Int

  insert                @IsOwner
  read    [descr,price] @PublicFollow
  update  [descr,level] @IsOwner

Follower
  user1   UserId
  user2   UserId
  status  Text

  assert                @OkFollows
  insert                @IsPending
  update  [status]      @OkOrNo

default read            @Allow
default insert, update  @Deny
```

```
declare follows : UserId → UserId → Bool

def IsOwner(row: Wish, user: User):
  row.owner == user.id

def Public(row: Wish, user: User):
  IsOwner(row, user) || row.level == "public"

def Follow(row: Wish, user: User):
  row.level == "follower" && follows(user.id, row.owner)

def PublicFollow(row: Wish, user: User):
  Public(row, user) || Follow(row, user)

def OkFollows(row: Follower):
  row.status == "ok" ⇒ follows(row.user1, row.user2)

def IsPending(row: Follower, user: User):
  row.user1 == user.id && row.status == "pending"

def OkOrNo(old: Follower, new: Follower, user: User):
  old.user2 == user.id && new.status `in` ["ok", "no"]

def Allow(row: a, user: User): True
def Deny(row: a, user: User): False
```

Figure 2: Refined Models: A centralized specification for the Wishlist App

```
showWishes user = do
  viewer   <- authUser
  let pub   = Level ==. "public"
  let chk   = if viewer == user then true else pub
  let qry = Owner ==. user &&. chk
  wishes <- select qry
  descrs <- mapM (project Descr) wishes
  respond (show descrs)
```

Figure 3: A `showWishes` controller. The highlighted code is needed for conformance with the `Public` policy.

API function `select` to get all the DB `Wish` rows belonging to `user`. Next, it extracts the `description` column for each row by invoking the ORM API function `project` with the name of the desired field. Finally, the controller uses the view API function `respond` to send the descriptions to the session user.

Enforcement Recall that the policy `Public` stipulates that `descriptions` should only be visible to the `owner` unless the `level` is `"public"`. Indeed, the `showWishes` controller, sans the shaded parts, is dodgy as the current session user could be asking for someone *else's* wishes! STORM detects this error at compile time, by: (1) inferring that the `qry` will return all rows owned by `user`, (2) using the policy on `Descr` to determine that the `project`'s results depend on values that are *allowed* to be viewable only by `user` (unless marked `"public"`), and then (3) complaining that by calling `respond` the results can be observed by the *sessionUser* who may be different than `user`.

We can fix `showWishes` by modifying the query when `user` is different than *sessionUser*. The modifications are shaded in Figure 3. First, we use the view API's `authUser` function to get the current session (`viewer`), which we use to add a `chk` clause to the DB query. When the target `user` *is* the session user, the `chk` clause is the trivial query `true` (which holds of all rows). However, if the target `user` is different, then the `chk` clause stipulates that the `level` column be `"public"`. The type checker infers that `qry` returns all rows owned by the session user, but *only* the public rows of *other* users. Hence, the type checker determines that the subsequent data release via `project` and `respond` conforms to the `Public` policy.

3.3 Information Flow Control

Next, let's see how STORM lets the programmer enforce IFC policies that (1) span values *across* different rows and tables, and (2) restrict how data flows to *multiple* users who may be unknown at the point where the data is accessed.

Policy Let us add social capabilities to our application by letting users have *followers* with whom they can share their wishes. We model this notion as a many-to-many `Followers` relationship table and then add `"follower"` as a new possible level value. Now the access to a particular Wish depends on data residing in another row, in another table—a record existing in the Followers table. STORM lets the programmer specify this requirement simply by changing the read policy for descr and price to @PublicFollow which is defined on the right in Figure 2. The key insight to specifying such a cross-table policy is that the existence of a Follower record *witnesses* the *follows* relationship between two users. The refined-models in Figure 2 makes this notion manifest as follows. First, at the top, we **declare** the relationship as a binary predicate follows between two UserIds. Second, the line assert @OkFollows says that for each row of the Follower table, the follows predicate holds between user1 and user2 *if* the status is "ok". Third, we use the predicate to define the Follow policy that says that when a wish's level is restricted to "follower" then the viewer user must be a follower of the wish owner. Finally, we use Follower to define a new policy PublicFollow that governs who is allowed to read the descr and price fields. This new policy captures our informal requirement about the three levels of viewers: "public", "private" and "follower".

Controller Continuing with the social aspect of the application, a nice feature would be to send an email notification containing a user's (non-"private") wish list, to all of the user's followers, a few days before that user's birthday. Our application implements this feature in the notifyFriends controller in Figure 4. The code starts by selecting the list of non-private wishes and projecting out their descriptions into the list descrs. Next, we query the DB to determine the list of followers flwUsrs. Finally, we use sendMail containing the wish decriptions descrs to all the users in flwUsrs.

Enforcement In the first phase notifyFriends accesses sensitive information that should only be made available to a data-dependent set of users who are, at that point, still to be determined. However, STORM's models API tracks this fact by combining the semantics of the wshQ query with the read policy associated with Descr to infer that only the followers of user are *allowed* access to the results of the first sub-computation that creates descrs. In the second phase, STORM's models API tracks the semantics of the flwQ query to determine that flws is a set of valid follows-tuples, and hence, that each user in flwUsrs is a valid follower of user. In the final phase, the signature for sendMail in STORM's view API checks that all the recipients in flwUsrs have the right access, and hence verifies the controller. If the programmer forgot the Status ==. "ok" clause, type checking would fail as flws would contain pairs with pending status, and hence, flwUsrs would contain possible non-followers outside the set allowed access by the first phase.

3.4 Implicit Flow Control

Next, let's see how STORM prevents *implicit* IFC violations involving publicly viewable data that was generated *conditioned* upon data the recipient should not be privy to. Recall, from Figure 2, that each wish has a price that should only be read

```
notifyFriends user = do
  -- Get list of wishes
  let wshQ = Owner ==. user &&.
             Level <-. ["public","follower"]
  wishes  <- select wshQ
  descrs  <- mapM (project Descr) wishes
  -- Get list of followers
  let flwQ = User1==. user &&. Status ==. "ok"
  flws    <- select flwQ
  flwIds  <- mapM (project User2) flws
  flwUsrs <- select (UserId <-. flwIds)
  -- Notify followers
  sendMail flwUsrs (show descrs)
```

Figure 4: A notifyFriends controller. The highlighted code eliminates the IFC violation of the PublicFollow policy.

```
usersWithExpensiveWishes min = do
  let qry = Price ≥. min &&. Level ==. "public"
  wishes <- select qry
  users  <- mapM (project Owner) wishes
  respond  (show (nub users))
```

Figure 5: A usersWithExpensiveWishes controller: The highlighted code eliminates the implicit flow violating the PublicFollow policy. The nub function removes duplicates from a list.

per the PublicFollow policy, i.e., by everyone (if "public"), by followers (if "follows") or else, only by the owner. The code in Figure 5 implements a controller that shows the session user a list of all the users that have a wish whose price exceeds the min threshold. (For now, ignore the shaded code.) If a programmer is not careful, they may think this code conforms to the application's policy as it returns a list of wish owners and owner is a publicly viewable column governed by the **default** read @Allow policy.

Enforcement However (absent the shaded code) STORM is unimpressed, as the list of expensive wishes was obtained by conditioning over the sensitive price column. STORM's models API tracks that the qry accesses the Price field, and infers that the result of the DB computation select qry should only be observed by users that satisfy the PublicFollow policy. Thus, when responding to the session user on the last line, STORM reports an error as it cannot prove that the session user satisfies the PublicFollow policy. To fix the code we must restrict the Price comparison to the wishes that the session user is allowed to access, for example, to all "public" wishes, as shown by the shaded diff in Figure 5. Now, as detailed in § 5.1, the type checker uses the models API to track the semantics of qry to infer that the results of the select computation may be made available to *all* viewers, thus verifying that the code conforms to the application's centralized policy.

4 Brief Intro to Refinement Types & IFC

STORM is implemented using two foundational blocks: Refinement types (§ 4.1) and Compositional IFC (§ 4.2).

4.1 Refinement Types

Refinement types let the programmer decorate the source program's types with logical assertions from a decidable logic to specify *subsets* of values of the decorated type [56, 57]. For example, the non-negative integers can be specified as

```
type Nat = {v:Int | 0≤v}
```

Pre- and Post-Conditions The user can write pre- and post-conditions for functions by refining the input and output types of functions. For example, `sum` adds the integers $0 \ldots n$

```
sum :: n:Nat → {v:Nat | n≤v}
sum 0 = 0
sum n = let t = sum (n-1) in n + t
```

We assign `sum` a refined function type, comprising an *input* type (pre-condition) that says that the function should only be invoked on non-negative integers, and an *output* type (post-condition) that says the result is a non-negative integer lower-bounded by the input n. Refinement type checking proceeds be generating a *verification condition* (VC), a logical formula whose validity implies the program type checks [9, 39, 58–60].

Bounded Refinements Generic APIs require a means of abstracting over particular policies and invariants of individual applications. We do so using *bounded* refinements [61] which allow (1) abstracting over the refinements (like type variables `<A ...>` abstract over concrete types) and (2) constraining the refinements with which the variables can be instantiated (like subtyping bounds `<A extends ...>` constrain type instantiation). For example, we can type the function composition operator `compose f g x = f (g x)` as

```
compose :: (Cmp f g r) ⇒ (y:b → {v:c | f(y,v)})
                       → (z:a → {v:b | g(z,v)})
                       → (x:a → {v:c | r(x,v)})
```

$$\texttt{where}\ \mathit{Cmp}\ f\ g\ r\ \doteq\ \forall x,y,z.\ g(x,y) \Rightarrow f(y,z) \Rightarrow r(x,z)$$

In the above, f, g and r are (abstract) *refinement variables*. The specification says that `compose` takes as input two functions that respectively map their argument y (resp. z) to an output v that satisfies the assertion $f(y,v)$ (resp. $g(z,v)$), and returns as output a function that maps its input x to a value v that satisfies the assertion $r(x,v)$. The abstract refinements f, g and r are related by the refinement bound $\mathit{Cmp}\ f\ g\ r$ which states that r is the relational composition of f and g. The signature is generic and precise in that it abstracts over the concrete post-conditions established by the arguments to `compose` while still letting us characterize the semantics of the result. Further, the (Horn clause) structure of the bound ensures that type checking remains decidable. Thus, we can use an SMT solver to automatically verify

```
sum2 :: n:Nat → {v:Nat | n≤v}
sum2 = compose sum sum
```

by automatically inferring that the refinement variables f, g, and r can all be instantiated to the refinement $\lambda\, n\, v \rightarrow n \leq v$.

4.2 Compositional IFC

Next, we give a high-level overview of the method used by STORM to enforce IFC in a compositional manner.

Primitive Operations and Computations An *application* is a collection of request *handlers*. Each handler is the sequential composition of a set of primitive *operations* that either read from or write to the database or send results to some users. For example, consider the handler e_{14} illustrated in Figure 6 that is composed from the primitive operations $e_1, \ldots, e_4$ as:

$$e_{12}\ =\ \texttt{do}\ e_1 ; e_2 \quad e_{34}\ =\ \texttt{do}\ e_3 ; e_4 \quad e_{14}\ =\ \texttt{do}\ e_{12} ; e_{34}$$

Thus e_{12}, e_{34} and e_{14} are *computations* built from primitive operations using the sequential composition (`;`) operator.

Authorizees and Observers Each primitive operation either *reads* data, e.g., from the database, that only a subset of users, the *authorizees*, are allowed to view, or *writes* data, e.g., to the network, thus providing it to a subset of recipients, the *observers*. For example, suppose that in the handler in Figure 6, the operations e_1 and e_2 read sensitive data with authorizees $auth_1$ and $auth_2$ respectively. Similarly, assume that e_3 and e_4 write data to observers obs_3 and obs_4 respectively.

Information Flow Control requires that whenever some primitive operation e_i reads data that is restricted to authorizees $auth_i$, all *subsequent* operations e_j only write data to observers obs_j that are contained in $auth_i$. For example, the handler in Figure 6 respects the given security policy if

$$\begin{array}{ll} obs_3 \subseteq auth_1 & obs_4 \subseteq auth_1 \\ obs_3 \subseteq auth_2 & obs_4 \subseteq auth_2 \end{array} \tag{1}$$

To enforce IFC we could expand each handler out into its sequences of primitive operations and then do the inclusion checks, e.g., via symbolic execution [14]. Sadly, this approach runs aground when there is a combinatorial explosion of paths through the handlers, or with loops or recursion which generate infinitely many possible computations.

Compositional Enforcement STORM circumvents path explosion using a two-step compositional approach [42, 44, 62], where each computation e is typed as $\langle auth, obs\rangle$ where $auth$ (resp. obs) under-approximates (resp. over-approximates) the authorizees (resp. observers) of e. First, STORM assigns the primitive operations the types

$$\begin{array}{ll} e_1 :: \langle auth_1, \emptyset\rangle & e_3 :: \langle \bar{\emptyset}, obs_3\rangle \\ e_2 :: \langle auth_2, \emptyset\rangle & e_4 :: \langle \bar{\emptyset}, obs_4\rangle \end{array}$$

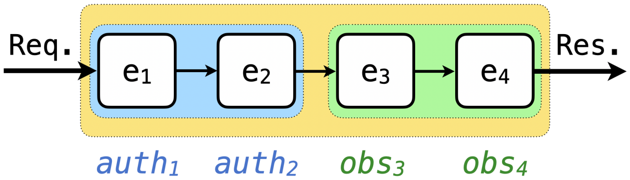

Figure 6: A request handler that sequences the primitive operations $e_1 - e_4$ with authorizees $auth_i$ and observers obs_j.

where $\emptyset$ and $\bar{\emptyset}$ are the empty and universal sets of users. Next, STORM assigns the ; operator a type that ensures that whenever we compose two computations e and e': (a) The observers of e' are *contained* in the authorizees of e, i.e., $obs' \subseteq auth$ (b) The authorizees of $e;e'$ are the *intersection* of those of e and e', i.e., $auth \cap auth'$, and (c) The observers of $e;e'$ are the the union of those of the sub-computations, i.e., $obs \cup obs'$. The implementations of e_{12} and e_{34} yield the (trivial) constraints $\emptyset \subseteq auth_1$ and $obs_4 \subseteq \bar{\emptyset}$, and types

$$e_{12} :: \langle auth_1 \cap auth_2, \emptyset \rangle \qquad e_{34} :: \langle \bar{\emptyset}, obs_3 \cup obs_4 \rangle$$

Finally, when we compose e_{12} and e_{34} to get the computation e_{14} we get the constraint $obs3 \cup obs_4 \subseteq auth_1 \cap auth_2$ which is equivalent to the IFC constraints (1). Next, let us see how our implementation represents the authorizees and observers with refinements and uses a typed API to compute, propagate and check those sets to enforce centralized security policies.

5 Implementation

We designed STORM to enable compile-time enforcement of centralized, data-dependent policies without any modification to the run-time. To achieve these goals, our design requires: (1) An expressive, data-dependent way to associate DB *fields* with the authorizees allowed access to those fields. (2) A way to connect DB *queries* with the authorizees allowed access to the query results. This set of users depends on the data in the underlying rows, so we also need to characterize the values of the rows returned by the query. (3) A way to *aggregate* the authorizees and observers across computations. (4) A way to ensure that observers who are *provided* sensitive data are a subset of the users authorized by the policy.

STORM achieves the above goals by *refining* the type abstractions (API) provided by each MVC layer with logical assertions that describe the *invariants* of the data processed by the operations, and the *policies* that govern access to that data. This is tricky as the assertions must simultaneously satisfy three properties. First, they must be *precise* to capture the semantics of the policies and DB operations. Second, they must be *generic* to enable reuse across many different web applications. Third, they must be *decidable* so applications can be automatically verified by SMT solvers. Next, we introduce the three principal data types of the STORM API (Figure 7) and use them to design a precise, generic and decidable API.

Policies A STORM *policy* is a binary predicate on a DB row and user, which we represent as a predicate of type `row → user → Bool`. The policy is data dependent as the predicate can use the `row`'s values to determine if a `user` is authorized. For example, Figure 2 specifies the policy *Public* as a predicate on the `Wish` row and a `user`. Each policy is attached to a *column* of a row specified in the ORM description in the `models` file. For example, in Figure 2 we attach the policy *Public* to the `description` column to specify that the description should only be viewable to users other than the `owner` when the row's access `level` is `"public"`.

Fields ORM libraries typically represent individual database columns as their own datatypes. STORM uses the PERSISTENT library [63] which represents each DB column as a type `Field row val` where `row` represents the underlying row (table), and `val` represents the value of the column itself. For example, in the code below, the DB table on the left is translated to the *fields* `Owner`, `Descr` and `Level` which respectively represent the corresponding DB columns as plain program values.

DB Table	ORM Fields
`Wish`	
`owner UserId`	`Owner :: Field Wish UserId`
`descr Text`	`Descr :: Field Wish Text`
`level Text`	`Level :: Field Wish Text`
`price Int`	`Price :: Field Wish Int`

Policies* in *Fields STORM's first pillar is a refined `Field` that represents policies at the type-level, by parameterizing the datatype with two abstract refinements (Figure 7):

$$pol:\ \texttt{row} \to \texttt{user} \to \texttt{Bool} \qquad sel:\ \texttt{row} \to \texttt{val} \to \texttt{Bool}$$

The refinement *pol* is instantiated with the policy attached to the `Field`; *sel* is a selector predicate that provides a type-level description of the value of the corresponding column. STORM uses the models file in Figure 2 to automatically generate the following types for `Owner`, `Descr`, `Level` and `Price`

```
Field ⟨⊥, λr v → v=r.owner⟩ Wish UserId
Field ⟨PublicFollow, λr v → v=r.descr⟩ Wish Text
Field ⟨⊥, λr v → v=r.level⟩ Wish Text
Field ⟨PublicFollow, λr v → v=r.price⟩ Wish Int
```

Thus, STORM's refined fields provide a uniform mechanism to lift data-dependent specifications up into types.

Queries Modern ORMs, going back at least to LINQ [64], allow the user to use `Field`s to build *queries*, e.g., of type `Query row` to represent query objects (or ASTs, not the results themselves) that access the DB table represented by `row`. STORM introduces a way to refine the types of the query API to track, at the type-level, the authorizees of the query results. As the policies are data-dependent, our API must also track the values of the rows in the query results. STORM achieves these goals via the second pillar of its API, a type that represents each DB `Query` parameterized by two refinements (Figure 7):

$$pol:\ \texttt{row} \to \texttt{user} \to \texttt{Bool} \qquad inv:\ \texttt{row} \to \texttt{Bool}$$

```
data Field ⟨pol: row → user → Bool, sel: row → val → Bool⟩ row val
data Query ⟨pol: row → user → Bool, inv: row → Bool⟩ row
data RIO   ⟨auth: user → Bool, obs: user → Bool⟩ val
```

Figure 7: The central types of the STORM API

As with `Field`, the refinement *pol* denotes the authorizees for each `row` returned by the query. Crucially, our query building API will ensure that *pol* intersects the authorizees across all the columns read by `Query`, not only those for the particular fields that are ultimately viewed by the viewers. This allows STORM to track implicit flows when filtering over sensitive columns, e.g., in the `usersWithExpensiveWishes` controllers from Figure 5. The refinement *inv* is an assertion that holds of every `row` returned by the query. The *inv* refinement enables type-level tracking of the query semantics which is essential for data-dependent policies. For example, the type

$$\texttt{Query}\ \langle \mathit{PublicFollow},\ \lambda r \rightarrow r.\mathit{level} = \text{“}\mathit{public}\text{”} \rangle\ \texttt{Wish}$$

describes a query on the `Wish` table, where (1) the query's results may only be accessed when the `level` is `"public"` or by the `owner`'s followers, and (2) each returned `row`'s `level` column has the value `"public"`.

Computations Standard ORMs use a monadic type to represent computations with side-effects. Haskell's `IO val` describes computations that access the DB, send email or network responses to yield a `val` value. The last pillar of STORM's API is the monadic `RIO` type that describes handlers and is parameterized with two refinements that track the authorizees and observers of the underlying computations (Figure 7):

$$\mathit{auth}:\ \texttt{user} \rightarrow \texttt{Bool} \qquad\qquad \mathit{obs}:\ \texttt{user} \rightarrow \texttt{Bool}$$

STORM ensures that in every `RIO` $\langle \mathit{auth},\ \mathit{obs} \rangle$ `val` computation (1) *auth* is an under-approximation of the authorizees of the data the computation depends upon, and (2) *obs* is an over-approximation of the observers to whom the computation provides data. STORM then prevents leaks by ensuring that when sub-computations e_1 and e_2 are sequenced, the observers of e_2 are contained in the authorizees of e_1 (§ 3.3).

5.1 Model API

STORM's models API lets applications compose `Field`s to build a `Query` and then to execute each `Query` to obtain an `RIO` computation that provides access to DB values (Figure 8).

Query Operators Standard ORMs let the programmer write atomic queries using relational operators that test whether the value of a column equals (or disequals, exceeds, etc.) some run-time program value. For example, `Level ==. "public"` in Figure 3 denotes a `Query` that will return all `Wish` rows whose `Level` column is `"public"`. Similarly, `Price` $\geq$`.` `min` in Figure 5 is a `Query` that will return all `Wish` rows whose price column exceeds the value of `min`.

Compile-time enforcement poses three challenges. First, the constructed `Query`'s type must track the policy describing the set of users who are allowed access to the `Field`s upon which the query result depends. Second, the constructed `Query`'s type must capture the invariant that each row returned by the query will, in fact, have the corresponding field-value equal-to `"public"`, or greater than `min`, etc. Finally, we must achieve the above in a generic fashion that abstracts over the underlying DB column, so that the programmer can reuse the operators like `==.` across different tables.

Refined Query Operators We solve the above challenges with the types for the refined query operators *equals* (`==.`), *not-equals* (`/=.`), *less-than* (`<=.`), *element-of* (`<-.`) in Figure 8. For example, the signature for the equality operator (`==.`) says that given (1) a `Field` indexed by a *pol*icy and *sel*ector, and (2) a comparison `value` satisfying a property p, the operator returns as output a `Query` with the same *pol*icy as the input `Field` where the resulting rows are guaranteed to satisfy the *inv*ariant. The crucial equality relationship is specified by the bound $\mathit{FldEq}\ \mathit{sel}\ \mathit{inv}\ p$ which says that

$$\forall r, \mathit{fv}, v.\ \mathit{sel}(r, \mathit{fv}) \Rightarrow p(v) \Rightarrow \mathit{fv} = v \Rightarrow \mathit{inv}(r) \qquad (2)$$

Recall that each `Field`'s *sel*-ector predicate characterizes the value of the `Field` in a given `row`. That is, $\mathit{sel}(r, \mathit{fv})$ holds when the value of the `Field` in `row` r is fv. Thus, the bound (2) says that for any `row` r, the invariant $\mathit{inv}(r)$ holds whenever the field's value fv equals any value v that satisfies p. To get a different comparison, e.g., less-than or disequality, we need only modify the $=$ relationship in the bound to $\leq$ or $\neq$ respectively.

Query Combinators ORMs let us use combinators to build complex queries from simpler ones. For example, the query `Level ==. "public" &&. Price` $\geq$`. min` in Figure 5 returns all `Wish` rows whose `Level` is `"public"` *and* `Price` exceeds `min`.

Compile-time enforcement requires the combinators' signatures meet two goals. First, the combined `Query`'s *pol*icy predicate should be the *intersection* of the users allowed access to each sub-query. Second, the combined `Query`'s *inv*ariant should be the conjunction (for `(&&.)`) or disjunction (for `(||.)`) of the sub-query invariants.

Refined Query Combinators We achieve the above with the signatures for `(&&.)` and `(||.)` in Figure 8. The conjunction combinator `(&&.)` takes two input sub-queries of type `Query` $\langle \mathit{pol}_1,\ \mathit{inv}_1 \rangle$ `row` and `Query` $\langle \mathit{pol}_2,\ \mathit{inv}_2 \rangle$ `row` respectively, and returns a `Query` $\langle \mathit{pol}_1 \sqcup \mathit{pol}_2,\ \mathit{inv} \rangle$ `row`. The output `Query`'s *pol*icy is the *join* of the two inputs, i.e., the set of authorized users is the intersection of those allowed by pol_1 and pol_2.

```
(==.) :: (FldEq sel inv p) ⇒ Field ⟨pol, sel⟩ row val → val⟨p⟩ → Query ⟨pol, inv⟩ row
  where
    FldEq sel inv p ≐ ∀r,fv,v. sel(r,v) ⇒ p(fv) ⇒ (fv = v) ⇒ inv(r)

(&&.) :: (And inv_1 inv_2 inv) ⇒ Query ⟨pol_1, inv_1⟩ row → Query ⟨pol_2, inv_2⟩ row → Query ⟨pol_1⊔pol_2, inv⟩ row
  where
    pol_1⊔pol_2 ≐ λr u → pol_1(r,u) ∧ pol_2(r,u)
    And p q r ≐ ∀x. p(x) ⇒ q(x) ⇒ r(x)

select :: (PolAuth pol inv auth) ⇒ Query ⟨pol, inv⟩ row → RIO ⟨auth, ⊤⟩ [row⟨inv⟩]
  where
    PolAuth pol inv auth ≐ ∀r,u. inv(r) ⇒ auth(u) ⇒ pol(r,u)

project :: (PolAuth pol inv auth) ⇒ Field ⟨pol, sel⟩ row val → row⟨inv⟩ → RIO ⟨auth, ⊤⟩ val
  where
    PolAuth pol inv auth ≐ ∀r,u. inv(r) ⇒ auth(u) ⇒ pol(r,u)

join :: (Auth_1 sel_1 sel_2 pol_1 inv auth, Auth_1 sel_1 sel_2 pol_q inv auth, Auth_2 sel_1 sel_2 pol_2 inv auth, SelOn sel_1 sel_2 on) ⇒
        Field ⟨pol_1, sel_1⟩ row1 val → Field ⟨pol_2, sel_2⟩ row2 val → Query ⟨pol_q, inv⟩ row1 →
        RIO ⟨auth, ⊤⟩ [(row1⟨inv⟩, row2)⟨on⟩]
  where
    SelOn sel_1 sel_2 on ≐ ∀r_1,r_2,v. sel_1(r_1,v) ⇒ sel_2(r_2,v) ⇒ on(r_1,r_2)
    Auth_1 sel_1 sel_2 pol inv auth ≐ ∀r_1,r_2,v,u. sel_1(r_1,v) ⇒ sel_2(r_2,v) ⇒ inv(r_1) ⇒ auth(u) ⇒ pol(r_1,u)
    Auth_2 sel_1 sel_2 pol inv auth ≐ ∀r_1,r_2,v,u. sel_1(r_1,v) ⇒ sel_2(r_2,v) ⇒ inv(r_1) ⇒ auth(u) ⇒ pol(r_2,u)
```

Figure 8: Selected functions from STORM's Models (ORM) API

The bound *And* inv_1 inv_2 *inv* states that the output `Query`'s *inv*ariant is the conjunction of that of the inputs' inv_1 and inv_2.

Example: Building Queries Let's see how STORM's API types the query `Level ==. "public" &&. Price ≥. min` from Figure 5. First, by composing the respective `Field` types for `Level` and `Price` with that of the `(==.)` operator, the type checker infers the left and right conjuncts have types

```
Query⟨⊥, λr→r.level = "public"⟩ Wish
Query⟨PublicFollow, λr→r.price ≥ min⟩ Wish
```

which `(&&.)` combines to type the conjoined query as

```
Query⟨PublicFollow, λr→r.level = "public" ∧ ...⟩ Wish
```

Selecting Rows Lastly, the API has functions to query the database. ORMs export a `select` function that executes a `Query` to return a list of matching `rows`. STORM's API refines the type of `select` to use the `Query`'s *pol*icy and *inv*ariant to determine: (1) the set of users authorized access to the results, and (2) the invariants of the result itself, as the data may then be used to generate subsequent queries. To this end, STORM assigns `select` the signature in Figure 8, which says that it takes as input a `Query` ⟨*pol*, *inv*⟩ `row` and returns as output a computation `RIO` ⟨*auth*, ⊤⟩ `[row`⟨*inv*⟩`]`. That is, the computation produces a list of `rows` where each row satisfies *inv*. The resulting computation's observers are the empty set $\top \doteq \lambda u \to false$. However, the computation's authorizees *auth* are defined by the bound *PolAuth pol inv auth* which says a user *u* is authorized to access a row *r* that satisfies the `Query` *inv*ariant *only when* that row and user satisfy the `Query` *pol*icy.

Projecting Fields In standard ORMs, the `rows` returned by `select` are opaque: a `project` operation must be used to extract the value of a given column (`Field`). STORM's API refines the type of `project` to track the authorizees of the extracted value via the signature in Figure 8, which says that `project` takes an input `Field` ⟨*pol*, *sel*⟩ `row val` and a `row`⟨*inv*⟩ and returns a computation `RIO` ⟨*auth*, ⊤⟩ `val`. Like `select` the computation has an empty set of observers (⊤). Further, the signature reuses `select`'s bound to ensure that computations authorizees *auth* are contained within those specified by `Field`'s *pol*icy.

Example: Selection and Projection Recall the `Query` in Figure 5 which looks for all the public `Wish` rows whose price exceeds `min`. As shown in the previous example, the `Query`'s *pol*icy and *inv*ariant predicates were inferred to be

$$pol \doteq PublicFollow \qquad inv \doteq \lambda r \to r.level = \text{“public”} \wedge \ldots$$

Thus, at the `select` the type checker infers the authorizees *auth* to be the set of *all* users, as the *inv*ariant implies the *pol*icy predicate. If, as in Figure 5, the `Level ==. "public"` clause was absent, the above implication would not hold, yielding a smaller set of authorizees *auth*. This would would render the handler ill-typed, as it (implicitly) leaks the sensitive `Price` value to observers outside *auth*.

Joining Tables ORMs let the user replace inefficient nested loops over multiple tables with efficient *join* operations.

STORM provides a `join` function that tracks (1) the authorizees of the sensitive data accessed by the query, and (2) the invariants of the resulting rows. STORM's `join` accounts for the policies in both tables via the signature in Figure 8. The type says that `join` takes as input the two `Field`s to join on (the `ON` clause) and a `Query` to filter the results (the `WHERE` clause), and returns a list of record pairs that satisfy the `Query`'s *inv*ariant and the *on* condition. The *on* condition is defined by the *SelOn* bound which says the condition holds for rows r_1 and r_2 if their respective join fields are equal. Further, the resulting computation's authorizees *auth* are defined by the bounds $Auth_1$ and $Auth_2$ which limits *auth* to users authorized to view the join and query fields for the subset of rows selected by the query.

Example: Join Recall the controller in Figure 4 which notifies the followers of a `user` after inefficiently computing them (`flwUsrs`) with two `select` queries: one to access the rows of the `Follower` table and one to get the corresponding rows of `User`. We can efficiently compute `flwUsrs` with a single `join`

```
let joinQ = User1==.user &&. Status==."ok"
flwUsrs  <- join User2 UserId joinQ
```

which returns a list of `(Follower, User)` pairs whose second component are `user`'s followers who can then be notified.

5.2 Controller & View API

Existing ORMs for effect-sensitive languages like Haskell encapsulate controllers and views in a monadic API to distinguish effectful DB and network computations from pure ones. STORM refines the monadic API to track the authorizees and observers of each controller computation.

Controller API The key element of the controller API is the monadic `bind` operator that sequences computations. When c_1 and c_2 are computations, of type `RIO a` and `RIO b` respectively, the expression `bind` c_1 $(\lambda x \rightarrow c_2)$ is the computation that runs c_1, *binds* its result of type `a` to `x` *and then* runs c_2. In Haskell and similar languages, sequential blocks

$$\texttt{do}\ \{x_1\ \texttt{<-}\ e_1;\ \ldots\ x_n\ \texttt{<-}\ e_n;\ e\}$$

are translated to

$$\texttt{bind}\ e_1\ (\lambda x_1 \rightarrow \ldots\ \texttt{bind}\ e_n\ (\lambda x_n \rightarrow e))$$

STORM's signature for `bind` (Figure 9) ensures three properties of any sequential composition `bind` c_1 $(\lambda x \rightarrow c_2)$. *(Leak-freedom)* First, we ensure that c_2 does not leak sensitive information accessed in c_1. That is, we ensure that the observers obs_2 of c_2 are contained in the authorizees $auth_1$ of c_1, via the bound $auth_1 \sqsubseteq obs_2$. *(Authorizee-strengthening)* Second, the the authorizees of the sequenced computation are $auth_1 \sqcup auth_2$: the users authorized to access the data read by *both* sub-computations. *(Observer-weakening)* Finally, the the observers of the sequenced computation are $obs_1 \sqcap obs_2$: the users who are observers of *either* sub-computation.

$$
\begin{array}{l}
\texttt{return} :: \texttt{a} \rightarrow \texttt{RIO}\ \langle\bot, \top\rangle\ \texttt{a} \\
\\
\texttt{bind} :: (auth_1 \sqsubseteq obs_2) \Rightarrow \\
\qquad\quad \texttt{RIO}\ \langle auth_1, auth_2\rangle\ \texttt{a} \rightarrow \\
\qquad\quad (\texttt{a} \rightarrow \texttt{RIO}\ \langle auth_2, obs_2\rangle\ \texttt{b}) \rightarrow \\
\qquad\quad \texttt{RIO}\ \langle auth_1 \sqcup auth_2, obs_1 \sqcap obs_2\rangle\ \texttt{b} \\
\quad \textbf{where} \\
\qquad auth \sqsubseteq obs \doteq \forall u.\ obs(u) \Rightarrow auth(u) \\
\\
\texttt{authUser} : \texttt{RIO}\ \langle\bot, \top\rangle\ \{u{:}\texttt{User} \mid u = sessionUser\} \\
\texttt{respond}\ \ : \texttt{Text} \rightarrow \texttt{RIO}\ \langle\bot, \lambda u \rightarrow u = sessionUser\rangle\ () \\
\texttt{sendMail} : [\texttt{user}\langle p\rangle] \rightarrow \texttt{Text} \rightarrow \texttt{RIO}\ \langle\bot, p\rangle\ ()
\end{array}
$$

Figure 9: Selections from STORM's Controller & View APIs

View API STORM's view API provides a function `authUser` whose signature (Figure 9) states that it returns the identity of the currently authenticated session user. Handlers can use this function to determine suitable responses to HTTP requests, e.g., by constructing and executing DB queries using `authUser` (§ 3.2). The view API has a `respond` function whose signature, shown in Figure 9, specifies that it takes `Text` or `JSON` data and sends it back to the currently authenticated *sessionUser*. Recall that the *Leak-freedom* guarantee provided by the type of `bind` ensures that whenever `respond` is used, the recipient is authorized to view the data used to construct the corresponding `Text` or `JSON` payload. Unlike previous frameworks which require potentially unsafe declassification [6], STORM's view API includes a way to `sendMail` responses to lists of users, where type checking ensures that data is disclosed per the application's centralized policy (§ 3.3).

5.3 Policies and Updates

Non-trivial applications require policies that relate rows *across* tables. (We found 9/11 of the benchmarks in our evaluation require policies that span tables § 7.1.) For example, in the `WishList` app (§ 3.3) we required that only the `owner`'s followers be allowed to read the `description` of a non-public `Wish`. The follower relationship is naturally stored in a separate `Follower` table. Hence, we must support policies that say that access is allowed if *there exists* a particular row in a different table. In the case of `WishList` a `user` can view a `descr` for a `Wish` when there exists a row in the `Follower` table whose `status` is `"ok"` that relates the viewer with the `Wish` owner. The direct way to specify such a policy is with *existentially quantified* refinement predicates, or alternatively to add a *relational join* to the set of logical operations. Unfortunately, both of these approaches take the predicate language out of the efficiently SMT decidable fragment, thus precluding automatic verification.

Witnessing Existentials with Predicates STORM allows cross-table policies by using uninterpreted predicates to provide evidence that certain rows exist in (other) tables. First, the policy `declare`s there is a suitable relation without providing any definition for it. For example, in Figure 2 we declare a binary `follows` predicate that holds for a pair of users. Second, the policy `assert`s that each record establishes the predicate holds for the tuple of values in the record. This predicate is then added as an *invariant* that holds of every record of the corresponding table. For example, in Figure 2 we `assert` that, e.g., `OkFollows` holds for each `Follower` record. Consequently, the type checker assumes that every term of type `Follower` satisfies the invariant, and hence, provides concrete evidence that the `follows` relationship holds between users in the record's fields, *if* the `status` is `"ok"`. In this way, STORM lets us specify cross-table policies, while ensuring refinements stay decidable.

Predicates vs. Updates Predicates are timeless: once the relationship is established it holds forever. This is problematic, e.g., if the record is updated or deleted, which would require us to similarly invalidate those invariants in the code. We reconcile the tension between timeless predicates and updates by separating two goals: (1) provide security guarantees *locally* within a single controller action, and (2) reflect the effects of updates and deletions *globally* across multiple controller actions. That is, locally, we want that within a single action, a `Alice` should be able to view `Bob`'s wishes only if at *some* point during the action the `Follower` table contained a tuple `(Alice, Bob, "ok")`. However, if during an action, `Bob` revoked access, e.g., by updating the `"ok"` to `"no"`, then in *subsequent* controller actions we must deny `Alice` access.

Soundness via Monotonicity and Erasure Our uninterpreted-predicate method achieves these goals as follows. First, we impose a syntactic restriction that the predicates appear positively (i.e., not under a negation). Implicitly, the predicates are interpreted to be true if they held of *any* database snapshot during the handler action. In other words, the predicates are *monotonic*: i.e., once established, they continue to hold till the end of the action. Second, STORM's compositional design *erases* the assertions at the end of each controller action, as each action is checked in isolation starting with no assertions. That is, the assertions must be re-established by future actions by querying the database, ensuring that if one action updates the database, e.g., to revoke privileges, then accesses will be prevented in subsequent handler actions. Thus, monotonicity lets us soundly enforce the policy locally in an action, and erasure lets us propagate the effects of updates globally across actions, essentially by viewing the predicates as holding *per handler action*.

6 Verification

We establish the security guarantees of STORM in two steps. First, we implement a formally verified Labeled IO (LIO) *library* [10], whose API ensures that well-typed *clients* do not throw dynamic IFC exceptions, i.e., do not leak. Second, we use our typed LIO library to implement λ_{STORM}, a simplified *reference implementation* of the STORM API. (Unlike λ_{STORM}, the full STORM implementation supports tables with arbitrary many columns and SQL types, and implements DB queries using existing ORM libraries backed by SQL databases.) As well-typed λ_{STORM} applications are well-typed LIO clients, we are guaranteed they do not leak.

IFC with Labeled Values In LIO, `Label`s are elements from a lattice whose partial order $\sqsubseteq$ specifies *allowed* flows [10]. LIO secures data by wrapping it with `Label`s indicating the level at which it is visible

```
data Labeled a = {val: a, lbl: Label}
```

LIO enforces IFC by maintaining an *ambient* (or *current*) label l_c which keeps track of the most sensitive value read during the computation. The ambient label l_c *starts* at $\bot$ and is *updated*, i.e., monotonically increased using the labels of the sensitive data accessed during the computation. The system enforces IFC by *blocking* any output to a security level *below* l_c, as this would correspond to an (undesirable) information flow from a high (e.g., `Secret`) level to a low (e.g., `Public`) level. The undesirable flow is blocked via a dynamic IFC exception that aborts the computation.

Refined LIO Computations LIO encapsulates secure computations in a *monadic* interface that systematically creates, propagates, updates labels to enforce IFC. To this end, LIO structures computations as *label-transformers* of type `LIO a` which are functions that take as argument the *current* label l and returns the *updated* label l' and the computation's result: a value of type `a`. λ_{STORM} refines `LIO a` to implement the computation type (§ 5) as

$$\texttt{type RIO } \langle auth, obs\rangle \texttt{ a =}$$
$$\{l{:}\texttt{Label} \mid l \sqsubseteq obs\} \to (\{l'{:}\texttt{Label} \mid l' \sqsubseteq l \sqcup auth\}, \texttt{ a})$$

The precondition requires that *obs* over-approximates the observers who are given access by the computation's ambient label l. The postcondition ensures that the updated label l' includes the authorizees for the computation.

Verified RIO API We make the `RIO` type abstract, and let developers write secure applications by exposing a monadic API (`bind` and `return`) extended with three operations. (1) `label l v` protects a value `v` by wrapping it with a label `l`. The operation enforces IFC by *checking* that the label `l` is not below the ambient label `lc`. If the check fails, the program `abort`s with a (dynamic) IFC error [10]. (2) `unlabel lv` takes a labeled value `lv` of type `Labeled a` and returns a computation producing the (unwrapped) `a` value. `unlabel` ensures the ambient label is updated at each sensitive data access by raising the ambient label to be at least that of `lv`'s label. (3) `downgrade l k` lets us safely unlabel `Boolean`-valued computations by taking *ceiling* label `l` and a `Boolean`-valued computation `k`, and then executes `k` *at* label `l`, updating the ambient label to `lc` $\sqcup$ `l`: Crucially, if the computation `k`'s label exceeds the ceiling `l`, then `downgrade` returns a *default* value `False`. This ensures that the `True` result is only observed for computations that safely occur *under* the ceiling `l`. We type the `RIO` API with refinements that verify

(a) the λ_{STORM} implementation of the API type-checks, and (b) well-typed clients do not throw IFC exceptions.

Policies For brevity, in λ_{STORM} we assume the DB stores a single type `Val` of primitive values and that each table has exactly two columns. In λ_{STORM}, a data-dependent *policy* is a function that maps DB rows' `Val`ues to `Label`s that protect access to each column

```
type Policy = Val → Val → Label
```

A `Spec` declares the policy for a table via one per column

```
data Spec = {p1:Policy, p2:Policy}
```

Tables A DB `Row` is a pair of labeled values

```
data Row = {f1:Labeled Val, f2:Labeled Val}
```

We define a type for `Row`s that are protected by the `Spec` s via the refinement *sat s r* which states that the row r's columns are labeled per s' policies

```
type RowS s = {r:Row| sat s r}
where sat s r ≐ ∧_{i∈1,2} s.p_i r.f_1.val r.f_2.val = r.f_i.lbl
```

Thus, we implement database `Table`s as a package

```
data Table = {spec: Spec, rows: [RowS spec]}
```

comprising a policy specification *spec*, and a collection of *rows* protected by labels satisfying *spec*. Thus, type checking ensures that every `Table` contains rows that are protected as mandated by the `Table`'s `spec`.

Verified ORM λ_{STORM} implements the models API (Figure 8) on top of our refined LIO interface in about 800 lines of code. We use `label` and `unlabel` to respectively implement `insert` and `project`. We implement `Query` using an algebraic datatype indexed with predicates that respectively represent the *pol*icy, and *inv*ariant associated with the query. Finally, we use `downgrade` to implement `select`, `update` and `join` and verify their correctness with a reference `eval` function that represents query semantics at the type-level. We use LIQUIDHASKELL to verify [65] that λ_{STORM} implements the API, which, coupled with previously established non-interference results for LIO [6, 10] proves λ_{STORM} applications do not leak.

7 Evaluation

We evaluate STORM by asking three questions: How *expressive* is STORM's policy specification mechanism? (§ 7.1) What typing *burden* does STORM's static verification place on developers? (§ 7.2) Does STORM *reduce* the code that developers need to get right in real applications? (§ 7.3)

7.1 Expressiveness

We evaluate the expressiveness of STORM's specification mechanism porting the *security policies* of nine case studies spanning four state-of-the-art approaches for centralized policy enforcement in web applications, summarized in Table 1: (i) From URFLOW [14] we ported a minimal application for storing `Secrets`; a message `Forum` with fine-grained access-control; a `Calendar` app where users share details of their schedule specifying who may learn details about it; and an anonymous `Poll` app where the creator can draft a poll and later mark it as live; (ii) From HAILS [66] we ported `GitStar`, a code hosting web platform inspired by GitHub; (iii) From JACQUELINE [5] we ported a `Conference` manager that supports designation of roles, paper submissions, assignment of reviews and review submissions; a `Course` manager that allows instructors and students to organize assignments and submissions; a `HealthRecord` Manager based on the HIPAA privacy standards; (iv) From LWEB [6] we ported `BIBIFI`, a web-site to host the "Build it, Break it, Fix it" security-oriented programming contest [67].

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	`gitstar`	16	21
LWEB	`bibifi`	312	101

Table 1: Expressiveness comparison: Numbers are LOC.

URFLOW's specification language is the closest to ours: policies are specified as declarative SQL queries over the DB state, instead of STORM's logical assertions. As such, we found porting URFLOW policies to STORM to be straightforward.

JACQUELINE uses multi-faceted execution to dynamically enforce policies specified as boolean functions. We were able to express all but one policy from the JACQUELINE case studies. The sole exception was a policy from the `Conference` manager where conflicts between PC members and papers are stored in a `PaperPCConflict` table. A PC member can only see the author and the content of a paper if there is *no* conflict present in this table. Our specification language does not support policies that depend on the *absence* of rows, and we thus have to express conflicts differently. Like in URFLOW, policies in STORM are limited to those that can be proven to hold issuing simple queries to the database, including joins, but without using more complex features like grouping or sorting rows, which we leave to future work.

HAILS and LWEB use labels to dynamically enforce policies. The policies in their case studies directly ported over to STORM. In many situations we were able to specify the requirements in a more natural and declarative way. Specifically, HAILS and LWEB accommodate data-dependent policies by querying the database at runtime to associate labels with meaning derived from the database state. For example, to even specify the `Follow` in the wishlist app (§ 3) one needs to query the database to check a corresponding `Follower` record exists. This is a problem. First, they duplicate DB queries as the data returned by these policy queries is often *also* relevant for the application logic. Worse, the queries may leak or fail, making it hard to reason about policy specification. In LWEB, such queries are *trusted* and written outside their declarative policy specification language. But even when they are *not* trusted (e.g., as in HAILS), exceptions in policy specification code due to failed (or unsafe) queries are hard to debug.

7.2 Effort

We evaluate the burden that STORM's static typing puts on the programmer by implementing three case-studies—`WishList` (§ 3) and the `Course` and `Conference` apps from JACQUELINE. We pick these because they have a wide range of policies that were previously thought to only be enforceable dynamically.

`WishList` (§ 3) allows users to save wishes and browse those of other users. We implemented a version with the `PublicFollow` policy which allows access to others' wishes when the wish is public or the user is a follower.

`Conference` [5] models a conference manager with a blind review process. Users can be authors of papers or PC members who write reviews. STORM enforces several policies: only a PC member should be able to view data that could reveal the identity of a reviewer; scores or the overall decision should be viewable by non-PC users only when the PC has made decisions public; even in the public stage, a paper's reviews should only be accessible to PC members or the papers' authors; some data like a paper's text should be visible to the PC or authors, but can be made public if the paper has been accepted.

`Course` [5] is a course management system with two kinds of users: students who enroll in courses, receive assignments and turn in submissions, and instructors who grade submissions and send final scores. STORM enforces policies like: only the instructor of the class or the student can view certain data like the student's final grade for the class; only the instructor or the authoring student can access an assignment submission.

Typing Annotations Static enforcement requires programmers to write some untrusted (and verified) type annotations. STORM uses the off-the-shelf LIQUIDHASKELL checker whose inference engine reduces the typing annotations needed for verification [68]. Hence, programmers need only annotate the *allows* and *gives* labels for top-level controllers with assertions describing the access provided by the controller. Many of these are *trivial* assertion where the computation (1) does not read or output sensitive data and may be typed `RIO` $\langle\bot,\top\rangle$ `a` or (2) is not composed with other sensitive computations and may be typed `RIO` $\langle\top,\bot\rangle$ `a`. The remainder express restrictions specified in policies, as exemplified by the signature for `Conference`'s `getReviews` controller:

$$p\!:\!\texttt{Paper} \;\rightarrow\; \texttt{RIO}\; \langle \lambda v \rightarrow \mathit{PcOrAuth}(v,p), \top\rangle\; \texttt{[Review]}$$

This says that user v can access the `Review`s of p only if v is on the PC or the decisions have been made and v authored p.

Quantitative Evaluation Table 2 summarizes our quantitative evaluation of the programmer effort needed for static enforcement. For each case-study, we show (1) the total lines of code of the application split across the client (where applicable), server, the DB model, and the policy specification; (2) the typing annotations required to statically verify that the server code conforms to the policy; and (3) the time taken to verify the application. Overall, our results show the programmer overhead is modest: 1 line of type (resp. non-trivial type) annotations every 19 (resp. 29) lines of code across the three case studies. We measured verification times using a commodity laptop running Arch Linux with 16GB of memory and a quad core Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz processor. While the results show room for improvement the times themselves were acceptable for interactive development: refinement type checking is modular and the developer focuses on one controller at a time, for which verification typically takes a few seconds.

7.3 Auditability

The ultimate proof-of-the-pudding is: *does* STORM *reduce the amount of code the developer has to get right in real web applications?* To answer this question, we built and deployed two new applications: VOLTRON and DISCO. In both applications, the code is divided into a browser-based *client* written using the VUE.JS framework [69] and a STORM *server* that handles and provides sensitive data. The client does not know anything about the security policies: all enforcement is done server-side, where the policies are used to statically restrict how data is provided in response to client requests.

VOLTRON allows instructors to simultaneously view the progress of multiple *groups* of students collaborating on in-class programming exercises. *Administrators* can create new *classes* and assign them an instructor. *Instructors* can then enroll students and assign them to groups. Each *group* is assigned a *hash* which gives them access to a text buffer that is synchronized in real-time using Google's `firebase` service [70], providing collaborative editing. While students can only access their group's buffer, instructors can view all their classes' buffers. VOLTRON has two essential policies: (1) only administrators can create classes and only instructors can enroll students to a class; and (2) a group's buffer is only accessible to the group's members and the class' instructor. We deployed VOLTRON for four month in Fall 2020 and regularly use it in two classes with about 50 and 100 students.

DISCO abbreviates *Distant Socialing*, an application that simulates the "hallway track" for facilitating social interaction in, e.g., a conference or workshop. In DISCO, an *organizer*

Application	**LOC**					**Ver. (s)**
	Server	*Models*	*Policy*	*Client*	*Annot.*	
`Conference`	644	25	57	-	43 (32)	79
`Course`	198	24	19	-	5 (1)	20
`WishList`	334	12	21	-	20 (12)	27
`Voltron`	756	32	37	1012	29 (17)	44
`Disco`	859	43	32	4630	43 (16)	120
Total	2851	140	166	5844	125 (72)	290

Table 2: Time (in seconds) to verify each application and lines taken by *Server* code, DB *Model* definitions, *Policy* specification code, *Client* code and typing *Annotations*. Non-trivial typing annotations are shown within parentheses.

can set up video chat rooms for *attendees* to join and talk to each other. Once logged in, attendees find themselves in the "Lobby" where they can see other users currently connected and view their "badges". Users can choose to "join" a room, in which case they enter a video chat with the other participants in that room. Organizers can limit the capacity of rooms and broadcast announcements to all users. Additionally, attendees can directly message each other. The majority of DISCO's policies correspond to some form of access control—e.g., operations like managing rooms and sending invitations are restricted to organizers, and personal details about individuals can only be edited by those users. We do, however, enforce two information flow policies: (1) only the recipient of a direct message is allowed to see its content; and (2) if a user has their visibility set to private, only people currently in their room can see their location.

DISCO was deployed at the Programming Languages Mentoring Workshop (PLMW) in June 2020 and at the Verification Mentoring Workshop (VMW) in July 2020. In the latter, we had about 107 registered users in all and a peak of 55 users using DISCO simultaneously. The application elicited very positive responses from users who wrote: "DISCO is great, it has been fantastic having it as a platform for social interactions at VMW!", "In my experience, DISCO worked amazingly well!", and "DISCO was among the best parts of VMW this year".

Quantitative Evaluation Table 2 compares the size of the policy specification code—that the developer has to get right—with the rest of the web application: the implementation of the server, and additionally the JavaScript clients for VOLTRON and DISCO. We find that for real applications like VOLTRON and DISCO, which require many controllers to implement the application functionality, STORM's policies account for under 4% of the server code, and under 1% if we include the client.

Discussion STORM helped discover an information flow bug in DISCO that arose due to the subtle interaction of two seemingly independent features—and would likely have gone unnoticed otherwise. First, DISCO users can set their *visibility* to private and the UI, accordingly, should not reveal to others when they join a room. Second, each DISCO room has an associated *topic* which is protected by a policy that allows users inside the room to change it. A type error alerted us to a conflict between these policies. In particular, enforcing the topic policy could implicitly reveal the location of an invisible user (violating the first policy). We designed and implemented VOLTRON without using explicit policies, and only added them afterwards. While the process of building VOLTRON took several person-months, the verification process required only minor changes to the code—including the checks that eliminated the implicit leak—and was finished in under two days. Our experience suggests developers informally consider policies when programming and structure code to facilitate verification.

8 Conclusion & Future Work

We presented the STORM framework for writing MVC-style web applications with statically enforced, data dependent security policies. STORM shows how the MVC architecture naturally lends itself to IFC, by centralizing policies as part of the model and then using a type-refined ORM API to track information flow across database queries and handler computations.

The `RIO` monad is the glue that binds together the different elements of STORM to precisely track the *effects*—each computation's authorizees and observers—needed to enforce IFC. In principle, it should be possible to integrate our approach to any language that supports similar fine-grained effect tracking. On the flip side, however, a limitation of our design is that programmers have to structure their controllers in the restricted `RIO` monad which limits the effects available to them. Our evaluation shows how a broad range of effects (database queries, HTTP requests, emails, random number generation) can be integrated into the `RIO` monad which sufficed to build real web applications. It would be interesting to investigate how to securely integrate other classes of effects (e.g., exceptions which are historically leaky).

Another limitation apparent from our models API is that it takes some toil to extend STORM to support DB operations like `select` or `join`, which restricts the DB queries the developer can write. In future work, it would be valuable to see how to support more expressive queries by designing a way to systematically and automatically refine an ORM library that supports a large fragment of SQL.

Acknowledgments

Many thanks to the reviewers and Geoff Voelker for providing excellent feedback on early drafts of this work. We are especially grateful to our shepherd Jon Howell for spending hours to help illuminate murky passages in the exposition. This work was supported by the NSF under grant no. CNS-1514435, CCF-1943623, CCF-1918573, CCF-1911213, CNS-2048262, and by generous gifts from Microsoft Research and Cisco.

References

[1] T. Bar, "Notifying our developer ecosystem about a photo api bug," 2018, https://developers.facebook.com/blog/post/2018/12/14/notifying-our-developer-ecosystem-about-a-photo-api-bug/.

[2] The OWASP Foundation, "OWASP Top Ten," 2020, https://owasp.org/www-project-top-ten/.

[3] ——, "Top 10 2013," 2013, https://wiki.owasp.org/index.php/Top_10_2013-Top_10.

[4] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications," *Journal of Computer Security*, vol. 25, 2017.

[5] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, "Precise, dynamic information flow for database-backed applications," in *PLDI*. New York, NY, USA: ACM, 2016, pp. 631–647.

[6] J. Parker, N. Vazou, and M. Hicks, "Lweb: information flow security for multi-tier web applications," *PACMPL*, vol. 3, no. POPL, pp. 75:1–75:30, 2019. [Online]. Available: https://doi.org/10.1145/3290388

[7] T. Armerding, "The IoT: Too big (and buggy) to patch?" 2018, https://www.synopsys.com/blogs/software-security/iot-big-buggy-patch/.

[8] D. Stefan, "LambdaChair policy," 2014, https://github.com/deian/lambdachair/blob/master/LambdaChair/Policy.hs.

[9] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. Peyton Jones, "Refinement types for haskell," in *ICFP*, J. Jeuring and M. M. T. Chakravarty, Eds. ACM, 2014, pp. 269–282. [Online]. Available: https://doi.org/10.1145/2628136.2628161

[10] D. Stefan, D. Mazières, J. C. Mitchell, and A. Russo, "Flexible dynamic information flow control in the presence of exceptions," *J. Funct. Program.*, vol. 27, p. e5, 2017. [Online]. Available: https://doi.org/10.1017/S0956796816000241

[11] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web application via automatic partitioning," in *SOSP*, T. C. Bressoud and M. F. Kaashoek, Eds. ACM, 2007, pp. 31–44. [Online]. Available: https://doi.org/10.1145/1294261.1294265

[12] B. J. Corcoran, N. Swamy, and M. Hicks, "Cross-tier, label-based security enforcement for web applications," in *SIGMOD*, 2009, pp. 269–282.

[13] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," *SOSP*, 2009.

[14] A. Chlipala, "Static checking of dynamically-varying security policies in database-backed applications," in *OSDI*, 2010.

[15] D. A. Schultz and B. Liskov, "IFDB: decentralized information flow control for databases," in *Eurosys*, Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, Eds. ACM, 2013, pp. 43–56. [Online]. Available: https://doi.org/10.1145/2465351.2465357

[16] M. Guarnieri, M. Balliu, D. Schoepe, D. Basin, and A. Sabelfeld, "Information-flow control for database-backed applications," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019, pp. 79–94.

[17] A. Sabelfeld and A. Myers, "Language-based information-flow security," 2003. [Online]. Available: citeseer.ist.psu.edu/article/sabelfeld03languagebased.html

[18] A. Sabelfeld and A. Russo, "From dynamic to static and back: Riding the roller coaster of information-flow control research," in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 2009, pp. 352–365.

[19] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama, "Faceted execution of policy-agnostic programs," in *PLAS*, 2013.

[20] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, "Laminar: Practical Fine-grained Decentralized Information Flow Control," in *PLDI*. ACM, 2009, pp. 63–74.

[21] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett, "All your ifcexception are belong to us," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 3–17.

[22] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," 2012.

[23] M. Ngo, N. Bielova, C. Flanagan, T. Rezk, A. Russo, and T. Schmitz, "A better facet of dynamic information flow control," in *Companion Proceedings of the The Web Conference 2018*, 2018, pp. 731–739.

[24] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 109–124.

[25] D. Volpano, C. Irvine, and G. Smith, “A sound type system for secure flow analysis,” *Journal of computer security*, vol. 4, no. 2-3, pp. 167–187, 1996.

[26] F. Pottier and V. Simonet, “Information flow inference for ml,” in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002, pp. 319–330.

[27] N. Broberg, B. van Delft, and D. Sands, “Paragon–practical programming with information flow control,” *Journal of Computer Security*, vol. 25, no. 4-5, pp. 323–365, 2017.

[28] N. Swamy, B. J. Corcoran, and M. Hicks, “Fable: A language for enforcing user-defined security policies,” in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 369–383.

[29] D. E. Denning and P. J. Denning, “Certification of programs for secure information flow,” vol. 20, no. 7, 1977.

[30] A. C. Myers, “JFlow: Practical mostly-static information flow control,” in *POPL*, 1999.

[31] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers, “Fabric: a platform for secure distributed computation and storage,” in *SOSP*. ACM, 2009.

[32] P. Buiras, D. Vytiniotis, and A. Russo, “HLIO: Mixing static and dynamic typing for information-flow control in haskell,” in *ICFP*, 2015, pp. 289–301.

[33] V. Rajani and D. Garg, “On the expressiveness and semantics of information flow types,” *Journal of Computer Security*, no. Preprint, pp. 1–28, 2019.

[34] M. Vassena, A. Russo, D. Garg, V. Rajani, and D. Stefan, “From fine-to coarse-grained dynamic information flow control and back,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019.

[35] B. Montagu, B. C. Pierce, and R. Pollack, “A theory of information-flow labels,” in *2013 IEEE 26th Computer Security Foundations Symposium*. IEEE, 2013, pp. 3–17.

[36] L. Lourenço and L. Caires, “Information flow analysis for valued-indexed data security compartments,” in *Trustworthy Global Computing*. Springer, 2014, pp. 180–198.

[37] ——, “Dependent information flow types,” in *Proceedings of the 42nd Symposium on Principles of Programming Languages*. ACM, 2015, pp. 317–328.

[38] N. Swamy, J. Chen, and R. Chugh, “Enforcing stateful authorization and information flow policies in Fine,” in *ESOP*, 2010.

[39] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, “Secure distributed programming with value-dependent types,” in *ICFP*, 2011.

[40] S. Chong, K. Vikram, and A. C. Myers, “Sif: Enforcing confidentiality and integrity in web applications,” in *USENIX Security*, 2007.

[41] E. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. Schneider, “Logical attestation: an authorization architecture for trustworthy computing,” in *SOSP*, 2011, pp. 249–264.

[42] P. Li and S. Zdancewic, “Encoding information flow in haskell,” in *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society, 2006, p. 16. [Online]. Available: https://doi.org/10.1109/CSFW.2006.13

[43] D. Schoepe, D. Hedin, and A. Sabelfeld, “Selinq: tracking information across application-database boundaries,” in *ICFP*, J. Jeuring and M. M. T. Chakravarty, Eds. ACM, 2014, pp. 25–38. [Online]. Available: https://doi.org/10.1145/2628136.2628151

[44] N. Polikarpova, D. Stefan, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama, “Liquid information flow control,” *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, pp. 105:1–105:30, 2020. [Online]. Available: https://doi.org/10.1145/3408987

[45] M. N. Krohn, “Building secure high-performance web services with OKWS.” in *USENIX Annual Technical Conference (ATC), General Track*, Jun. 2004.

[46] A. P. Felt, M. Finifter, J. Weinberger, and D. Wagner, “Diesel: applying privilege separation to database access,” in *Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 416–422.

[47] R. Cheng, W. Scott, P. Ellenbogen, J. Howell, and T. Anderson, “Radiatus: Strong user isolation for scalable web applications,” University of Washington, Tech. Rep., 2014.

[48] A. Blankstein and M. J. Freedman, “Automating isolation and least privilege in web services,” in *Security and Privacy*. IEEE, 2014, pp. 133–148.

[49] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, “Breakapp: Automated, flexible application compartmentalization,” in *NDSS*, 2018.

[50] A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel, “Qapla: Policy compliance for database-backed systems,” in *USENIX Security Symposium*. USENIX, 2017, pp. 1463–1479.

[51] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan, “Building web applications on top of encrypted data using Mylar,” in *NSDI*, 2014, pp. 157–172.

[52] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun, “Verena: End-to-end integrity protection for web applications,” in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 895–913.

[53] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song, “Shadowcrypt: Encrypted web applications for everyone,” in *CCS*, 2014, pp. 1028–1039.

[54] D. Muthukumaran, D. O’Keeffe, C. Priebe, D. M. Eyers, B. Shand, and P. R. Pietzuch, “Flowwatcher: Defending against data disclosure vulnerabilities in web applications,” in *CCS*, I. Ray, N. Li, and C. Kruegel, Eds. ACM, 2015, pp. 603–615. [Online]. Available: https://doi.org/10.1145/2810103.2813639

[55] F. Wang, R. Ko, and J. Mickens, “Riverbed: Enforcing user-defined privacy constraints in distributed web services,” in *NSDI*. Boston, MA: USENIX, 2019, pp. 615–630.

[56] R. L. Constable and S. F. Smith, “Partial objects in constructive type theory,” in *LICS*, 1987.

[57] J. Rushby, S. Owre, and N. Shankar, “Subtypes for specifications: Predicate subtyping in PVS,” *IEEE TSE*, 1998.

[58] P. Rondon, M. Kawaguchi, and R. Jhala, “Liquid types,” in *PLDI*, 2008.

[59] J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis, “Refinement types for secure implementations,” in *CSF*, 2008.

[60] J. Hamza, N. Voirol, and V. Kuncak, “System FR: formalized foundations for the stainless verifier,” *PACMPL*, vol. 3, no. OOPSLA, pp. 166:1–166:30, 2019. [Online]. Available: https://doi.org/10.1145/3360592

[61] N. Vazou, A. Bakst, and R. Jhala, “Bounded refinement types,” in *ICFP*, K. Fisher and J. H. Reppy, Eds. ACM, 2015, pp. 48–61. [Online]. Available: https://doi.org/10.1145/2784731.2784745

[62] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke, “A core calculus of dependency,” in *POPL*. ACM, 1999, pp. 147–160.

[63] M. Snoyman and G. Weber, https://www.yesodweb.com/book/persistent.

[64] M. Torgersen, “Querying in c#: how language integrated query (LINQ) works,” in *OOPSLA*, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds. ACM, 2007, pp. 852–853. [Online]. Available: https://doi.org/10.1145/1297846.1297922

[65] R. Jhala and N. Lehmann, github.com/storm-framework/core.

[66] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, “Hails: Protecting data privacy in untrusted web applications,” in *OSDI*, C. Thekkath and A. Vahdat, Eds. USENIX Association, 2012, pp. 47–60. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/giffin

[67] A. Ruef, M. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel, “Build it, break it, fix it: Contesting secure development,” in *CCS*, 2016, pp. 690–703.

[68] B. Cosman and R. Jhala, “Local refinement typing,” *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, Aug. 2017. [Online]. Available: https://doi.org/10.1145/3110270

[69] “Vue.js: The progressive javascript framework,” https://vuejs.org/.

[70] “Firebase realtime database,” https://firebase.google.com/docs/database.

A Artifact Appendix

Abstract

Our artifact contains (a snapshot of) the source code for the implementation of STORM from § 5, the formally verified reference implementation λ_{STORM} described in § 6), the various policies, case-studies and applications used in our evaluation § 7.

Scope

The artifact provides a way to reproduce the results in the paper. First, we provide examples of how a programmer might write insecure code that fails to respect particular policies, as described in Section 3 and show how those mistakes are caught by refinement type checking. Next, the code shows how STORM is implemented on top of existing ORM and networking libraries as described in § 5. Further, the artifact contains the verified reference implementation of λ_{STORM} from Section 6 which shows how the API can be implemented on top of an LIO interface. Finally, addition, to the source code described above we include the various scripts used to compile the applications and measure the verification time and code annotation overheads that we report in Section 7.

Contents

The artifact comprises the following sub-directories and files: `storm-core`—the source for the verified reference implementation λ_{STORM} (§ 6); `models`—the ported policies from the expressiveness benchmarks (§ 7.1); `case-studies`—the source for the ported case-studies (§ 7.2); `disco` and `voltron`—the source for the end-to-end applications (§ 7.3); and `fig9.py`—the script used to generate Table 2. Each sub-directory contains a manifest file that links to the `github` commits for STORM and LIQUIDHASKELL that are needed to compile the application.

Hosting

You can obtain the artifact from `github` by running `git clone --recursive` on the repository `https://github.com/storm-framework/artifact` It suffices to use the `main` branch, specifically, commit `3eb138ab5145e688504eff71c669c6570701e10b`.

Requirements

You can run the artifact on any machine computer running Linux or MacOS after installing the following software. The artifact requires `python 3.7` and the following dependencies. (1) `stack v2.5.1` which can be installed by following these instructions[1]; (2) `z3 v4.8.8` which can be installed by downloading the binary [2]. You can ignore the shared libraries and bindings for Java and Python; just download and place a suitable `z3` binary somewhere in your `PATH`. (3) `tokei v12.1.2` which is used to count lines of code [3]. Familiarity with the `stack` build system for Haskell would be useful to evaluate the artifact but it is not necessary.

λ_{STORM} Implementation (§ 6)

Directory `storm-core` has the source for the verified reference implementation λ_{STORM} from § 6. To verify, run `cd storm-core && stack build`.

Policies (§ 7.1)

The code in `models/` contains the policies ported to evaluate expressiveness as described in § 7.1. This directory does not contain verifiable code, only the ported models files. The models files are grouped by the original tool they were taken from, e.g., the models file for the `Calendar` application in URFLOW is in `models/src/UrWeb/Calendar/Model.storm`.

[1] https://docs.haskellstack.org/en/stable/README

[2] https://github.com/Z3Prover/z3/releases/tag/z3-4.8.8

[3] https://github.com/XAMPPRocky/tokei#installation

Case Studies (§ 7.2)

The case studies used to evaluate the burden STORM puts on programmers as described in § 7.2 are in `case-studies`. There is a `stack` project for each case study.

Verify the Code To verify one of the case studies go to the corresponding directory and build the project. For example, to verify the `WishList` application run `cd case-studies/wishlist && stack build`.

Breaking the Code To check how STORM catches leaks open `case-studies/wishlist/src/Controllers/Wish.hs`. The function `getWishData` at line 156 extracts the information out of a `Wish`. The query between lines 164 and 171 checks if the viewer is friends with the owner of the wish. Remove the check `frienshipStatus ==. "accepted"` from the query, i.e., the query should look like:

```
friends  <- selectFirst
  ( friendshipUser1' ==. owner &&:
    friendshipUser2' ==. viewerId )
```

Then verify by running `stack build`. Forgetting to check if the friendship is `"accepted"` causes a leak as the viewer may not be friends with the `Wish` owner, yielding an error:

```
    |
173 | level == "friends" ->
    |   project wishDescription' wish
    |           ^^^^^^^^^^^^^^^^
```

Automation Evaluation (Fig 2)

To produce the count of lines of code in 2 run `python3 fig9.py`

Application: Disco (§ 7.3)

Verify the Code To verify `Disco`'s server code is leak free run `cd disco/server && stack build`

Break the Code Open the file `disco/server/src/Controllers/Room`. The function `updateTopic` on line 36 implements the functionality that allows a user to update a room's topic. If not done carefully, this operation may produce a subtle information flow bug as described in the discussion of § 7.3. `Line 42` checks that the user's visibility is set to `"public"` and only then allows them to update the topic. Update lines 42 to 50 to

```
Just roomId -> do
  UpdateTopicReq {..} <- decodeBody
  validateTopic updateTopicReqTopic
  _ <- updateWhere
        (roomId' ==. roomId)
        (roomTopic' `assign` updateTopicReqTopic)
  room   <- selectFirstOr notFoundJSON
            (roomId' ==. roomId)
  roomData <- extractRoomData room
  respondJSON status200 roomData
Nothing -> respondError status403 Nothing
```

and run `stack build`. Forgetting to check if the visibility is set to public produces an error when accessing the user's current room as the information may be leaked. You should see:

```
**** LIQUID: UNSAFE ***********************
src/Controllers/Room.hs:39:23: error:
...
   |
39 |   userRoom <- project userRoom' viewer
   |                       ^^^^^^^^^
```

Application: Voltron (§ 7.3)

Verify the Code You can verify the code by `cd voltron/server && stack build`

Break the Code Open the file `voltron/server/src/Controllers/Class.hs` The function `addRoster` at line 102 implements the functionality to enroll a list of students to a class. This operation is restricted to instructors of the class which is checked by the query in lines 109 and 110. Removing the clause `classInstructor' ==. instrId` so the query reads:

```
cls <- selectFirstOr
        (errorResponse status403 Nothing)
        (className' ==. rosterClass)
```

produces an error as the user does not have enough permissions:

```
**** LIQUID: UNSAFE ************************
src/Controllers/Class.hs:113:19: error:
...
   |
113| mapT (addGroup clsId) (rosterGroups r)
   |               ^^^^^

src/Controllers/Class.hs:114:19: error:
...
   |
114| mapT (addEnroll clsId) (rosterEnrolls r)
   |                ^^^^^
```