Scooter & Sidecar: A Domain-Specific Approach to Writing Secure Database Migrations

John Renner
UC San Diego, USA

Alex Sanchez-Stern
UC San Diego, USA

Fraser Brown
Stanford, USA

Sorin Lerner
UC San Diego, USA

Deian Stefan
UC San Diego, USA

Abstract

Web applications often handle large amounts of sensitive user data. Modern secure web frameworks protect this data by (1) using declarative languages to specify security policies alongside database schemas and (2) automatically enforcing these policies at runtime. Unfortunately, these frameworks do not handle the very common situation in which the schemas or the policies need to evolve over time—and updates to schemas and policies need to be performed in a carefully coordinated way. Mistakes during schema or policy migrations can unintentionally leak sensitive data or introduce privilege escalation bugs. In this work, we present a domain-specific language (Scooter) for expressing schema and policy migrations, and an associated SMT-based verifier (Sidecar) which ensures that migrations are secure as the application evolves. We describe the design of Scooter and Sidecar and show that our framework can be used to express realistic schemas, policies, and migrations, without giving up on runtime or verification performance.

CCS Concepts: • **Security and privacy → Information accountability and usage control**.

Keywords: database migration, verification, secure ORM, domain-specific language

ACM Reference Format:
John Renner, Alex Sanchez-Stern, Fraser Brown, Sorin Lerner, and Deian Stefan. 2021. Scooter & Sidecar: A Domain-Specific Approach to Writing Secure Database Migrations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454072

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454072

1 Introduction

Protecting user data in modern database-driven Web applications is hard: web developers must ensure that their application code correctly safeguards user data with every request. Unfortunately, popular Web frameworks don't help developers get this right because they don't account for security policy. They rely on developers to implement ad-hoc security mechanisms—to properly sprinkle just the right `if` statements throughout their code. When developers inevitably fail, their applications expose sensitive user data—from credentials [27] to COVID-19 test results [48] to user data from previously hacked databases [25].

Model-policy-view-controller (MPVC) frameworks (e.g., Lifty [40], Jacqueline [64], LWeb [39], and Hails [20, 21]) promise to address this problem by making data-access policies first class. MPVC frameworks allow developers to specify access-control policies on data alongside their schemas—the code describing the data model and interfaces used to access data—often using a declarative domain-specific language (DSL). For example, when defining the schema for user profiles, developers might specify a policy like "only the user can modify their profile and only they and their friends can see their email address." MPVC frameworks enforce such policies automatically, for example, when controllers—the code handling user requests—access the data model. This reduces the amount of code developers need to get right [20, 39]: instead of correctly implementing thousands of checks, they simply need to write correct, declarative policy code.

Reality, though, is more complicated than traditional MPVC frameworks suggest. Both models and policies constantly evolve through *migrations*. And unsafe schema or policy migrations can have devastating consequences—from leaking sensitive data to privilege escalation attacks.

Schema migrations modify and extend data models. When performing schema migrations, developers must (1) correctly reconcile their changes to the schema with changes to the application code (e.g., the controllers that interface with the new data modes and the view code that renders the data to, say, HTML or JSON) and (2) ensure that their changes are secure and do not violate the policy used to safeguard existing data. This is hard because migrations are typically written in low-level, error-prone database interfaces—most

often, directly in SQL—rather than the high-level object relational mappers (ORMs) used in application code [62, 63]. And even if developers manage to write correct migrations (e.g., using synthesis to automatically update the application code according to the new schema [63]), they still need to ensure that their migrations abide by the policy. In practice this is manual and error prone [14, 28]: migration tools are not aware of policies, so developers must manually ensure their code abides by the policy, with no room for error—unsafe migrations (e.g., copying sensitive data to public locations) often do not break application functionality and thus go unnoticed until it's too late (e.g., the sensitive data is leaked).

Policy migrations can also introduce security vulnerabilities when extending or modifying data-access policies. Existing frameworks make no guarantees about a new policy's relationship to the old policy, which can result in users gaining access to sensitive or critical data that would otherwise have stayed safe. Such leaks were common enough in Hails (e.g., in the Task management app [49]) that the authors modified their policy DSL with a new keyword that (tried to) make it clear when a field policy was updated to be publicly accessible [20]. This isn't unique to Hails or MPVC frameworks, though: it happens in traditional MVC frameworks, too. For example, the authors of the Ghost blogging platform unintentionally allowed contributors to edit blog posts [18], and they introduced this bug in a patch *itself* designed to fix a bug in their policy code [19]. Likewise, a refactor of HotCRP's policy code [30] inadvertently granted unauthenticated users administrator rights [29].

We address unsafe migrations via three contributions.

1. The Scooter Domain-Specific Language. Our first contribution is a new domain-specific language, Scooter, that allows developers to (1) declaratively specify data models and security policies on these models; and (2) write imperative schema and policy migrations, which update the data models and policies. Today, developers use wildly different languages for these tasks (e.g., ORM for describing the data model, custom DSL for policy specification, and SQL for schema migrations) and, as discussed above, ensuring that changes to models and policies are safe is a manual, error-prone task. By unifying specification and migration, Scooter makes it easier for developers to write safe migrations.

2. The Sidecar Verifier. Our second contribution is a static tool, Sidecar, which verifies the safety of migrations written in Scooter. At its core, Sidecar relies on an automated procedure that determines whether one policy is as strict or stricter than another. To verify a data migration, we use a static abstract analysis to track the flow of information across the migration and use this procedure to verify that the policy used to safeguard migrated data is at least as restricting as the policies on the sources of that data. Similarly, to verify a policy migration, we use the automated procedure to compare the new policy against the old. We designed Scooter and Sidecar together to ensure that verification is fully automatic and fast. Moreover, when verification fails, we designed Sidecar to generate a counterexample to help developers understand and debug policy violations.

3. Implementation and Evaluation. Our third contribution is an implementation and evaluation of the Scooter DSL and the Sidecar verifier. We implemented Scooter and Sidecar in Rust. For migrations that pass the verifier, the Scooter compiler generates (1) a *migration interpreter* that performs the verified-safe migration, (2) an authoritative specification containing the declarative model and policy, and (3) a typed Rust ORM implementation for each model. The generated ORM enforces policies automatically at run time and forces developers to update their application code to account for schema changes "for free", i.e., by generating the ORM we ensure that schema changes manifest as type errors. We evaluate Scooter and Sidecar on seven case studies from the MPVC literature, including a port of LWeb's Build it Break it Fix it, as well as a Ruby-on-Rails application used at UC San Diego for PhD Visit Day. We find that: (1) Scooter can express almost all policies and migrations from these previous efforts; (2) Scooter's ORM policy enforcement imposes under 11% overhead, which is comparable to previous work [20, 39]; and (3) Sidecar verifies most safe migrations in under a second, and for unsafe ones (e.g., the HotCRP migration from [29]) it generates useful counterexamples.

2 Motivation and Overview

In this section, we give a brief overview of how migrations—both traditional database migrations and updates to policy code—can introduce security vulnerabilities and how Scooter eliminates these vulnerabilities.

The Chitter MPVC Application. We use a simple social media web application, Chitter, as an example. Chitter allows users to post 42-character messages—peeps—on a public bulletin board. Though peeps are public, the app also handles sensitive information about users, e.g., follower relationships, private messages, pronouns, email addresses, and passwords. The Chitter developers are serious about protecting this data and use an MPVC framework to (1) separate the model and policy code from the rest of the application (the views and controllers) and (2) enforce the policy code automatically, at runtime. Figure 1 gives part of Chitter's data model for user profiles and its policy. The policy states that a user's email address is only visible to that user and Chitter administrators (who are, themselves, users); that the user's pronouns are visible to the user and the users they follow; and that the user and any admin can modify all but the `isAdmin` field, which can only be modified by admins.

```
User
  name: String
   read: public
   write: u -> [u] + User::Find({isAdmin: true})
  email: String
   read: u -> [u] + User::Find({isAdmin: true})
   write: u -> [u] + User::Find({isAdmin: true})
  pronouns: String
   read: u -> [u] + u.followers
   write: u -> [u] + User::Find({isAdmin: true})
  isAdmin: Bool
   read: u -> [u] + User::Find({isAdmin: true})
   write: u -> User::Find({isAdmin: true})
  followers: Set(User)
   read: u -> [u] + u.followers
   write u -> [u] + User::Find({isAdmin: true})
  ...
```

Figure 1. Chitter users model and policy in (simplified) Scooter.

2.1 Unsafe Migrations

Though using an MPVC framework helps the Chitter developers safeguard user data at runtime, modifying and extending the model or policy code could undermine this effort.

Unsafe Schema Migrations. Extending model schemas can inadvertently leak data and allow users to bypass data access policies. Consider, for example, changing the Chitter application by extending the user model with a new public `bio` field. To do so, Chitter developers modify their model:

```
User
...
+ bio: String
+  read: public
+  write: u -> [u] + User::Find({isAdmin: true})
...
```

They also write a migration—in SQL—that extends the underlying database to populate the new `bio` field, in this case with the user's name and pronouns:

```
ALTER TABLE user ADD bio STRING;
UPDATE user
SET bio = CONCAT("Hi! I'm ", user.name,
                 "(", user.pronouns, ").")
```

Since this migration isn't constrained by a policy, it *accidentally leaks sensitive data*—the pronouns—to all users.

Direct leaks like this are not the only concern, though. Migrations can modify data used by policy code—and unintentionally introduce leaks or privilege escalation bugs, i.e., grant users access to data they otherwise would not be able to access. For example, setting the `isAdmin` field of a user allows that user to read and write other users' profiles.

The problem is that *schema migrations are decoupled from policy code*—so developers must implicitly and informally enforce their data access policy on each migration. Doing so for Chitter is easy, but real-world migrations and policies are far more complicated.

Unsafe Policy Migrations. Changes to policy code can also introduce leaks and privilege escalation bugs. Consider further extending the Chitter application with a new hierarchy of administration: admins and moderators. Moderators, unlike admins, should only be allowed to read (and edit) free-form data like names and bios, which could contain potentially inappropriate content. To add support for moderators, the Chitter developers replace the boolean `isAdmin` field with an integer, `adminLevel`—where 0 is used for normal (unprivileged) users, 1 for moderators, and 2 for admins. They then update the model and policy in several steps.

They start by extending the user model with `adminLevel`:

```
User
...
+ adminLevel: Int
+  read:  u -> [u] + User::Find({adminLevel: 2})
+  write: u -> User::Find({adminLevel: 2})
...
```

Then, they perform a schema migration to add the new field, setting the admin level according to the old `isAdmin` field:

```
ALTER TABLE user ADD adminLevel INT;
UPDATE user
SET adminLevel = CASE WHEN isAdmin
  THEN 2
  ELSE 0
END;
```

Next, they update the policy code to use `adminLevel` instead of `isAdmin` and only then remove `isAdmin` from the model and underlying database (via another migration).

The new policy is introduced as an edit to the old:

```
User
...
  email: String
-  read: u -> [u] + User::Find({isAdmin: true})
+  read: u -> [u] + User::Find({adminLevel: 2})
-  write: u -> [u] + User::Find({isAdmin: true})
+  write: u -> [u] + User::Find({adminLevel: 2})
...
  bio: String
-  write: u -> [u] + User::Find({isAdmin: true})
+  write: u -> [u] + User::Find({adminLevel >= 0})
...
```

Alas, this policy is overly permissive: instead of restricting the bio field writers to the user, moderators, and admins, the new policy accidentally allows *any user* to write.

The problem is that *policy migrations are decoupled from policy enforcement.* Therefore, the burden is on developers to ensure that migration code doesn't sidestep the declared data

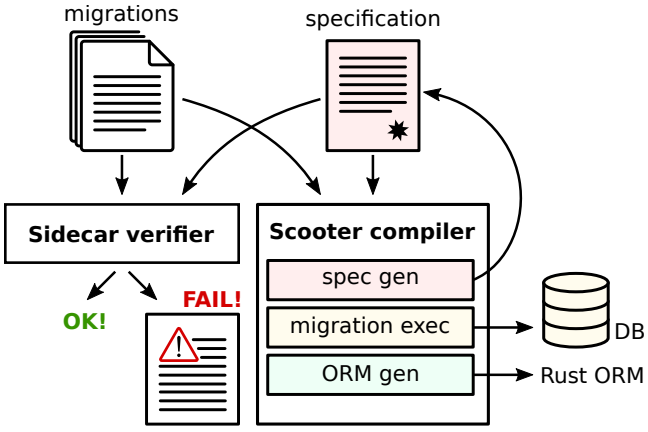

Figure 2. Given a migration and data model and policy specification, Sidecar verifies the safety of the migration. If the migration is unsafe, Sidecar produces a counterexample. Otherwise, the Scooter compiler (1) updates the specification, (2) executes that migration against the database, and (3) generates a type-safe policy-enforced Rust ORM, which appliations use to access persistent data.

access policies. Unsurprisingly, developers get this wrong—and when they do, the error is silent. Changes that inadvertently weaken policies persist until someone is lucky enough to notice them [29] or loud enough exploiting them [55].

2.2 Safe Migrations with Scooter and Sidecar

With Scooter, developers don't directly modify models and policies, nor do they write schema migrations using low-level interfaces like SQL. Instead, they use Scooter for both tasks (Figure 2). To start, developers implement a migration that generates the initial model and policy specification; Figure 1 gives an example of a (simplified) specification generated by Scooter.[1] Then, all policy and schema migrations are relative to and update this specification (and its underlying database representation). Let's consider how the Chitter migrations would be implemented using Scooter.

Preventing Unsafe Schema Migrations. With Scooter, the Chitter developers extend the original user model with bios by writing a migration script:

```
1 User::AddField(
2  bio : String {
3   read: public,
4   write: u -> [u] + User::Find({isAdmin:true})
5  }, u -> "I'm "+u.name+"("+u.pronouns+")");
```

This script extends the `User` model with a new string field `bio` and populates the bio according to the anonymous function on line 5. Our Sidecar verifier, however, catches the leak before it's too late: Sidecar automatically infers that `u.pronouns` data ends up in `u.bio` and that the `bio` policy is less restrictive than the `pronouns` policy. Scooter will only execute the migration after the developers modify the update function to not use `u.pronouns`.

Preventing Unsafe Policy Migrations. To extend Chitter with moderators, the Chitter developers, again, write a single script (instead of multiple scripts and file edits):

```
1 User::AddField(
2  adminLevel : I64 {
3   read: u -> [u] + User::Find({adminLevel:2}),
4   write: u -> User::Find({adminLevel: 2})
5  }, u -> if u.isAdmin then 2 else 0);
6
7 User::UpdateFieldPolicy(email, {
8  read: u -> [u] + User::Find({adminLevel: 2}),
9  write: u -> [u] + User::Find({adminLevel: 2})
10 });
11 User::UpdateFieldWritePolicy(bio,
12  u -> [u] + User::Find({adminLevel >= 0}));
13
14 User::RemoveField(isAdmin);
```

This migration would add the new `adminLevel` field, update the email and bio policies accordingly, and remove the old `isAdmin` field. Sidecar, however, catches the unsafe policy update on lines 11–12, stops Scooter from executing the migration, and generates a counterexample showing the policy violation:

```
Principal: User(0)
# CAN NOW ACCESS:
User { id: User(1),
       isAdmin: false,
       adminLevel: 0,
       bio: "",
       ...  }
# OTHER RECORDS:
User { id: User(0),
       isAdmin: false,
       adminLevel: 0,
       ...  }
```

This, in effect, forces the Chitter developers to fix the unsafe policy. They could do this by rewriting the policy to:

```
u -> [u] + User::Find({adminLevel: 2}),
```

This policy is safe—it is equivalent to the old one—but it's not the intended policy. The intended policy *is* more permissive: it should allow the user, admins, *and moderators* to edit `bios`.

Scooter allows developers to weaken policies but requires them to be explicit about the change being more permissive:

```
User::WeakenFieldWritePolicy(bio,
  u -> [u] + User::Find({adminLevel > 0}),
  "Reason: allow moderators to update bios.");
```

Being explicit does not prevent developers from getting such migrations wrong (e.g., using >= instead of >). It does, however, make it easier to audit migrations and narrow the focus of security reviews.

[1] As we describe in Section 3, Scooter is slightly more verbose. For example, developers need to specify who is allowed to create and delete objects, and not just who is allowed to read and write fields.

Safe Migrations Workflow. As Figure 2 shows, once Sidecar verifies the safety of a migration, Scooter performs the actual database migration and updates the model specification to reflect any model or policy changes. The model (and policy) specification is then used to generate a Rust ORM library, which allows application code to retrieve and modify persisted objects using a standard high-level typed interface (e.g., `User::Find`) that automatically enforces data-access policies. Like previous work [39], generating ORM code from the DSL specification also forces developers—via normal compiler type errors—to update the view and controller code to account for schema changes. We describe Scooter in more detail next.

3 Design

We built the Scooter languages according to three main design goals:

1. **Unification**. Right now, developers typically use different languages to manage their databases (e.g., SQL) define their policies (e.g., Hails), and query the database (e.g., ORMs). Scooter aims to provide a unified semantics for migrations and policy specification that mirrors popular ORM patterns.
2. **Expressiveness**. A unified language is only effective if it can express the union (or more!) of what separate languages are able to. For example, the language must be able to express both real-world policies and real application data models.
3. **Verifiability**. The verifier must catch safety violations statically, with informative and actionable error messages.

Scooter maintains a single *policy file*, written in Scooter_p, that contains the current schema and all policies (unification). Users *do not* manually update this file or write Scooter_p directly. Instead, Scooter automatically updates the file when users write and run Scooter_m migration scripts—the scripts that make schema changes, manipulate data, and update new policies. Before executing a migration, Scooter verifies that the migration's changes are safe with respect to the current policy (verifiability). This process is illustrated in Figure 2.

In this section, we elaborate on how Scooter fulfills our three design goals. First, we explain how schemas and policies are expressed in Scooter_p (§3.1). Then, we explain how users express changes to the schema and policy through migrations in Scooter_m—and how, as a result, Scooter is able to statically prevent migration errors (§3.2). Finally, we describe how users write applications using Scooter's ORM (§3.3).

3.1 Declaring Policies

Scooter expresses both schemas and policies because they are inherently coupled; policies guard access to the data defined in the schema and are themselves defined in terms of queries against that same schema.

Schemas. Scooter uses a standard ORM data model, defining schemas in terms of *models* composed of typed *fields*. Figure 4 shows a policy file containing a simple `User` model with a single `name` field of type `String`.

To express relational data, Scooter generates an implicit `id` field that acts as a unique identifier for each instance—i.e., database row—in the model. This allows one model instance to hold a reference to another (e.g., the `bestFriend` field in Figure 4 refers to the user `id`). The `id` field is strongly typed—`User::id` is of type `Id(User)`, allowing type-safe object lookups.

While `id`s are powerful enough to express any relational construct—one-to-one, one-to-many, many-to-many—they cannot express all *policies* on their own (§6.3). Scooter's `Set` type allows users to express otherwise inexpressible policies and, moreover, makes it easy to express simple one-to-many relations. For example, a `User` may have many emails (`Set(String)`).

Principals and Policies. In Scooter, applications operate on data through basic create, read, update, and delete (CRUD) *operations*, where each operation is performed on behalf of a *principal*. *Policies* define, for each operation, the set of principals allowed to perform that operation. For example, Figure 4 states that all principals can read a `User`'s name.

What is a principal? In Scooter, many principals are simply database object `id`s. To specify that a model's `id` is a valid principal, Scooter annotates that model with `@principal`. For example, the `User` in Figure 4 is a principal and is annotated as such. We call this kind of principal a *dynamic principal* because its existence is tied to the state of the database.

Sometimes, though, it's important to express policies in terms of application infrastructure (as opposed to database objects). Scooter uses *static principals* for this purpose. For example, Figure 4 declares an `Unauthenticated` static principal, which the application can use for operations not made on behalf of a logged-in user. The policy also states that the `Unauthenticated` principal is the only one allowed to create `User`s; in other words, only users who are not logged in are able to create new users. While the set of static principals varies by application, we find two to be very common: an `Unauthenticated` principal and a `Login` principal that has read access to all password data but is used sparingly.

Policy Functions. Scooter expresses the relationship between operations and principals (e.g., `User`s and their ability to change their usernames) as a *policy function* from the target instance of the operation (e.g., `User`s) to a set of principals allowed to perform that operation (e.g., change usernames). For example, Figure 4 states that the policy for writing to `User::name` is `u -> [u.id]`. Scooter uses square brackets to denote sets, so this function says: "for any user u, the set of principals allowed to change its name contains only

(variable) $var ::= x_0, x_1, .., x_n$
(datetime) $datetime ::=$ now | d<month>-<day>-<year>-<hour>:<minute>:<second>
(constant) $const ::= string, integer, float, datetime,$ true, false, public
(binary ops) $binop ::= gencmp \mid op \mid numcmp$
$op ::=$ + | -
$numcmp ::=$ < | <= | > | >=
$gencmp ::=$ == | !=
(find operators) $fop ::=$: | ∋ | $numcmp$
(set literal) $set ::= [e_0, .., e_n]$
(functions) $func ::= var$ -> e
(expressions) $e ::= const \mid set \mid var \mid !e \mid (e\ binop\ e)$
| (if e then e else e) | (match e as var in e else e) | None | Some(e)
| e.map($func$) | e.flat_map($func$) | $e.field$
| $Model$::ById(e)
| $Model$::Find({$field_1\ fop_1\ e_1, \ldots, field_n\ fop_n\ e_n$})

Figure 3. The syntax of value expressions shared between policies and migrations in Scooter.

```
@static-principal
Unauthenticated

@principal
User {
  create: _ -> [Unauthenticated],
  delete: none,

  name: String {
    read: public,
    write: u -> [u.id]},
  bestFriend: Id(User) {
    read: u -> [u.id, u.bestFriend],
    write: u -> [u.id]},
  adminLevel: I64 {
    read: public,
    write: u -> User::Find({adminLevel: 2})
                  .map(u -> u.id)}}
```

Figure 4. Simple user profile and principal declaration in Scooter.

u.id". The language contains two convenience terms for common functions: public, which is a function that returns all principals, and none, which is the same as _ -> [].

For any operation on a model m, the type of a policy function must be $m \rightarrow$ Set(Principal). Within the function, the policy is free to traverse instances and query the models to construct its output set: Scooter policies use conditionals, mathematical operations, comparisons, and comparison-based querying.

Policy functions are strongly typed expressions. This ensures that policies cannot crash at runtime—they will always produce a set of principals—and simplifies the lowering of policy expressions to SMT (for verification). We give the full type system for policies in Appendix B.

3.2 Migrations

Users update their policies and schema by writing migration scripts in Scooter$_m$. Migration scripts consist of a series of commands that modify the schema (e.g., add a field) and the policy. Crucially, Scooter$_p$ and Scooter$_m$ share an underlying semantics, unlike traditional MPVC frameworks where policies are expressed with models and migrations are expressed in raw SQL. Differing semantics make migration safety verification hard, while unified semantics—as with Scooter$_p$ and Scooter$_m$—make verification easy.

Schema Changes. In Scooter, users can change schemas by creating and deleting models or by creating or deleting fields of those models. Whenever users create a model or field, they must include all read, write, create, and delete policies. Consider the following migration, which extends the policy in Figure 4 with public posts called Peeps:

```
1  CreateModel(Peep {
2    create: public,
3    delete: p -> [p.author],
4
5    author: Id(User) {
6      read: public,
7      write: none,
8    },
9  });
10 Peep::AddField(body: String {
11     read: public,
12     write: p -> [p.author],},
13 p -> "Peep by " + User::ById(p.author).name);
```

This migration first creates a Peep model containing an author (lines 1–9), then adds a peep body (lines 11–14). Line 14 specifies that all peeps receive a default body that states the author's name. This is *required* in Scooter: when developers

add a field to a model, they must provide a function that populates that field with an initial value.

Migrations can also remove fields and models as long as other policy functions do not depend on them. For example, the following would fail, because the body policy above refers to `author`:

```
Peep::RemoveField(author);
```

On the other hand, the following would work, because no policies (other than those within `Peep`) depend on `Peep`:

```
DeleteModel(Peep);
```

Policy Changes. In addition to schema updates, Scooter migrations can express policy updates. For example, a developer may want to update the `create` policy on `Peep`:

```
Peep::UpdatePolicy(create, p -> [p.author]);
```

This migration replaces the previous policy (`public`) with a new policy function that only allows users to create a peep when they are the author; previously, anyone could create a peep with any author. The `UpdatePolicy` command indicates the developer's intent is to provide a policy that is at least as strict as the old policy; the verifier will prove that this is true before Scooter executes the migration. If developers need to weaken a policy, they can use `WeakenPolicy`, and the verifier won't check for strictness preservation. Finally, developers can also strengthen and weaken field policies with `UpdateFieldPolicy` and `WeakenFieldPolicy`.

Principal Changes. Scooter migrations can also change the set of principals using `AddPrincipal`, `RemovePrincipal`, `AddStaticPrincipal`, and `RemoveStaticPrincipal`, which have no effect on the underlying schema. The verifier will stop developers from removing any principal that is used in policy functions.

3.2.1 Verifying Migrations.

Scooter verifies the safety of an entire migration before it executes any part of it, which obviates rolling back migrations partway through because of errors. The main challenge for verification is that the correctness of one migration command depends on its predecessors. For example, consider the following migration, which creates a `User` model and then adds a `bestFriend` field and a `secret` field that is shared between a user and their best friend.

```
1  CreateModel(User {
2    create: public,
3    delete: u -> [u.id],
4  });
5  User::AddField(bestFriend: Id(User) {
6    read: public,
7    write: u -> [u.id],
8  }, u -> u.id);
9  User::AddField(secret: String {
10   read: u -> [u.id, u.bestFriend],
11   write: u -> [u.id],
12 }, _ -> "my_secret");
```

The `User::AddField` commands are only valid after the `User` model has been created (using `CreateModel` on line 1). If lines 1–4 were omitted, Scooter would reject the migration because of a missing `User` model. Likewise, the `read` policy for `secret` does not typecheck unless the `bestFriend` field has already been added to `User`.

To address the fact that each migration command's safety depends on prior commands, Scooter maintains an in-memory representation of models models that it uses to typecheck and verify each command. Once Scooter has verified a command, it records the command's effect on the set of models and continues on to verify the next command until the migration is complete or it hits an error. Scooter does not actually manipulate data during this process, so in case of an error, Scooter doesn't need to rollback database state. When Scooter has verified the migration completely, it executes the migration against the database and writes the in-memory policy to the Scooter_p file.

3.3 The Scooter ORM

Following a successful migration, the Scooter compiler generates an ORM implementation in Rust for each model. The ORM enforces policies dynamically before performing database queries. Like most ORMs, our ORM is agnostic to the underlying database system and relies on a *driver* to communicate with an actual database; we implement and evaluate a MongoDB driver. Since our ORM is largely standard, we only describe the Scooter-specific details.

Acting on Behalf of Principals. Before querying an ORM model, developers must declare a principal with which to perform the query. For example:

```
1 // set up the db connection
2 let db_conn = // ...
3 // declare the principal
4 let princ = db_conn.as_princ(Unauthenticated);
5 // query the database
6 let u = User::find_by_id(princ, some_user_id);
```

Web applications rarely require manual principal management, though: typically, middleware automatically selects a principal based on the signed-in user (e.g., instead of the `Unauthenticated` principal on line 5).

Handling Overly Sensitive Fields. Queries to the database return *partial objects*; the ORM removes fields that the principal does not have read access to. In turn, developers must handle fields whose values are missing due to policy enforcement. For example, in this code snippet, they must account for a missing `email` field:

```
match u.email {
  // principal has read permissions:
  Some(email) => println!("Success");
```

```
    // principal does not have read permissions:
    None => println!("Failure");
}
```

This forces developers to explicitly consider permissions. In practice they need to do this already, for example, when implementing views.

Handling Policy Failures. When writing to the database, the ORM checks the relevant `create` or `update` policies and returns an error if necessary. For example, in this snippet, the ORM code accounts for an attempted edit that could fail because `princ` does not have the proper edit permissions:

```
match edited_user.save(princ) {
  Ok(_) => println!("Save successful")
  Err(_) => println!("Save failed"),
}
```

These errors force developers to respond to both policy and database failures. In development mode they can respond with the exact access violation; in production mode, they can simply return an HTTP 403 Forbidden response.

4 Verifying Policy Updates in SMT

The core of our Sidecar verifier is centered around proving that one policy is at least as strict as another, a property we call *strictness*. This *strictness* property not only allows Sidecar to verify `Update` commands, but also allows it to prevent data leaks. In this this section, we first formalize the strictness safety property and describe how Sidecar translates this property into SMT formulas, allowing it to verify `Update` commands using an off-the-shelf SMT solver—specifically, Z3 [13]. Finally, we show how Sidecar uses strictness to detect leaks.

Strictness Property. Recall that a Scooter policy (for a given operation) is a function p that takes an instance, i.e., an object, and returns the set of principals who are allowed to perform that operation. Because policies can query the database, p must also take the database as a parameter. Formally, it is safe to strengthen policy p_1 to policy p_2 iff the following strictness property holds:

$$\forall db, \forall i \;.\; p_1(db, i) \supseteq p_2(db, i) \tag{1}$$

That is, for all databases and for all instances (objects) in those databases, p_2 must produce a subset of the principals returned by p_1.

For each migration of a policy from p_1 to p_2, Sidecar checks the migration's safety by translating this formula into an SMT query. Unfortunately, a direct translation of this formula to SMT leads to many different problems, some related to performance and some related to counterexample generation (which requires the solver to generate a full database). To sidestep these issues, Sidecar translates policies into set-free SMT queries. We describe our translation to SMT next.

Leakage Formula. SMT solvers verify a property by proving that its negation is unsatisfiable. When we negate the strictness property we get the core of our SMT query:

$$\exists db,\ i,\ u.\ u \in p_2(db, i) \wedge \neg(u \in p_1(db, i)) \tag{2}$$

That is, there exists some database db, instance i, and principal u, such that u is permitted by p_2 and not by p_1. While we can express this in SMT directly (using the theory of arrays [34]), our translation eliminates sets and set comparison. First, it translates set fields to an equivalent join-table representation. Next, Sidecar distributes the $\in$ operator across all expressions, to eliminate all remaining set-typed expressions and variables. We describe these next.

Translating Set Fields. The Scooter language allows users to define fields that contain sets. For example, a single user on a social media site may have many followers, which programmers express in Scooter as follows:

```
User { ... friends: Set(Id(User)) { ... } }
```

Standard ORM practice translates the above into a join table. We adopt a similar approach—we encode the user-friend relation explicitly by adding the following model of the relation:

```
UserFriends { from: Id(User) {...},
              to:   Id(User) {...} }
```

Using this encoding, `friends` field access can be translated into an appropriate query on the `UserFriends` table. The Sidecar verifier performs this translation at the language level, before translating to SMT. For example, it translates the expression `user.friends` into:

```
UserFriends::Find({from: u}).map(uf -> uf.to)
```

Translating Set Expressions. Once set fields are removed, Sidecar rewrites the leakage condition (2) into an equivalent formula without sets. Sidecar does this by distributing the $\in$ operator across Scooter expressions. In most cases this is straightforward. For example, $u \in (e_1 + e_2) \rightsquigarrow (u \in e_1) \vee (u \in e_2)$. The two exceptions are `map`, `flat_map`, and `Find`.

When Sidecar distributes $\in$ across `map` it introduces an existential:

$$u \in e_1.\texttt{map}(x \rightarrow e_2) \rightsquigarrow \exists v.\ v \in e_1 \wedge u = e_2[v/x]$$

Similarly for `flat_map`:

$$u \in e_1.\texttt{flat\_map}(x \rightarrow e_2) \rightsquigarrow \exists v.\ v \in e_1 \wedge u \in e_2[v/x]$$

Because all instances used in policies are in the database, when translating $u \in M\texttt{::Find}(\{\ldots\})$, we can simply check if u meets the criteria of the `Find` query:

$$u \in M\texttt{::Find}(\{\ \ldots\ f_i\ op_i e_i\ \ \ldots\}) \rightsquigarrow \bigwedge_i (u.f_i\ op_i\ e_i)$$

This translation eliminates all remaining set expressions and variables from the leakage formula. We give the complete definition of $\rightsquigarrow$, as well as proofs of correctness and set elimination, in Appendix A.

Translating Instances and IDs. Sidecar translates instances to SMT by encoding each field as a function, much like Nijjar et. al [38]. For example, Sidecar translates the declaration `email: String` inside `User` into a function `email : User → String`. So, `u.email` in Scooter is translated to `(email u)` in SMT. Instances like `u` in our SMT encoding are uninterpreted values (which can only be used as parameters to field functions like the `email` function above). This encoding also allows us to easily encode id-uniqueness. In particular, instead of asserting uniqueness as $\forall o_1, o_2 \,.\, o_1.id = o_2.id \implies o_1 = o_2$, we define an `id` function (which represents the `id` field) to return the instance itself, i.e., we define `(id i)` to return `i`. This avoids expensive quantifiers and reduces $o_1.id = o_2.id$ to $o_1 = o_2$.

Translating Primitives. In addition to `Bool` which is trivially represented in SMT, Scooter supports integers (`I64`), doubles (`F64`), `Option` types, and `DateTime`. `I64` and `F64` are naturally represented as bitvectors whose operations are encodable in first-order logic and are built-in to Z3. We encode `Options` using SMT-LIB's datatype declarations. `DateTime` requires special handling. We encode `DateTimes` as UNIX timestamps (bitvectors in SMT). This allows Scooter to use integer bitvector comparison to implement `DateTime` comparison. Sidecar models the `now()` constructor as an unconstrained bitvector. When comparing two policies, Scooter assumes they are invoked at the same time and thus uses the same unconstrained value for all occurrences of `now()`.

Detecting Data Leaks. Sidecar uses the policy strictness check, combined with dataflow analysis, to detect data leaks. We say data leaks when, during migration, data flows from a more restrictive field to a more permissive field. For example, this (leaky) migration moves data from the private email field to a public bio:

```
CreateModel(User {
  create: public,
  delete: u -> [u.id],

  email: String {
    read: u -> [u.id],
    write: u -> [u.id],
  }
});
AddField(bio: String {
  read: public,
  write: u -> [u.id],
}, u -> u.email);
```

Before the migration, a user's email was only visible to the user; afterwards, everyone can read the email since it is used to initialize the bio field.

We detect leaks in two steps. First, we use a simple static *dataflow* analysis on the Scooter language to detect which fields flow to which other fields during the migration. Second, for each field f_1 that flows to a field f_2 we check that the policy for f_2 is at least as strict as the policy for f_1.

Using Prior Definitions. Sometimes the correctness of a policy migration relies on the schema migrations that preceded it. To reiterate the example from Section 2.1:

```
User::AddField(
  adminLevel : I64 {
    read: _ -> User::Find({adminLevel: 2})
    write: _ -> User::Find({adminLevel: 2})
  }, u -> if u.isAdmin then 2 else 0);
User::UpdateFieldWritePolicy(bio,
 u -> [u] + User::Find({adminLevel: 2}));
```

While Sidecar normally encodes fields as uninterpreted functions in SMT, in this example Sidecar proves this migration safe by using a *prior definition*. Specifically, the initialization function used for `adminLevel` defines the relationship between `adminLevel` and `isAdmin`—so Sidecar knows that a user has an `adminLevel` of 2 if and only if `isAdmin` is true. Using prior definitions is necessary for verifying certain migrations, but can also have surprising semantics (§6.4).

5 Evaluation

We evaluate our Scooter language, our Sidecar verifier, and our policy-enforcing Rust ORM by answering the following questions:

1. Can the Scooter language express common policies and migrations (§5.1)?
2. Can Sidecar detect unsafe migrations? (§5.2)
3. Is the Sidecar verifier performant enough to use regularly (§5.3)?
4. Is the overhead of the generated ORM in line with existing policy enforcement techniques (§5.4)?

To answer these questions we port case studies from existing policy frameworks to Scooter: one from LWeb [39], three from Hails [21], one from Lifty [40], and one from UrFlow [8]. We use MPVC case studies because they contain explicit policies. None of these case studies provide migrations, however, so we reconstruct them, when possible, from git histories. In addition, we port a production Ruby on Rails application—and its migrations—to Scooter. Rails does not provide a policy language, so we reverse engineer policies from the behavior of the application.

Results Summary. We find that Scooter is capable of expressing the vast majority of policies and migrations showcased by existing frameworks—confirming that Scooter's verification-oriented design decisions don't unduly limit the expressiveness of the language. We find that Sidecar can detect unsafe migrations (and generate counterexamples) from real applications. Furthermore, we find that verification with Sidecar takes less than a second to complete and that our ORM imposes a runtime overhead (11%) comparable to the LWeb [39] and Hails [20] MPVC frameworks.

Experimental Setup. We conduct all performance measurements on an Arch Linux (kernel 5.11.9) desktop with an Intel i7 6700K processor, 4 cores (8 hyperthreads) at 4GHz, and 16 GB of RAM.

5.1 Scooter Language Expressiveness

To answer this question we port case studies from existing policy frameworks to Scooter and discuss the capabilities and limitations of the language. For each case study, we report the number of migrations, lines of migration code, the number of policies successfully ported, and the number of migration actions used. Our results—reported in Figure 5—show that Scooter is able to represent all policies and all but one migration. In the remainder of this section, we discuss each case study and the process of porting them to Scooter.

Build it Break it Fix it. We port the LWeb BIBIFI production web application designed to manage and coordinate security programming contests [56]. The application allows administrators to create contests and posts related to those contest and manage the teams and scores of contests, while regular users can log in and see the current contests and their team's details. LWeb developers write policies on fields of a record as disjunctions of static principals and (other) fields of the record. BIBIFI uses automatic schema migrations to (1) remove fields or (2) add fields with default values. Scooter is able to express all the BIBIFI data models, policies, and migrations.

Visit Day. We port a production Ruby on Rails app designed to schedule meetings between visiting prospective PhD students and faculty [43]. The application allows users to create privileged accounts for scheduling meetings, as well as unprivileged accounts so that students and faculty can view their schedules; users can reset their passwords and invite other users. We port both the application and its ActiveRecord [17] migration scripts with no issues. There are ten migrations, totaling 139 lines of code. The original hand-written policy is 25 lines of code, but after all the migrations, it becomes an automatically generated 103-line policy file.

GitStar. We port the GitStar [57] benchmark—a lightweight GitHub-like application—from the Hails project [21]. We make one modification. Hails repositories have a `reader` field that can be a set of user ids or a special `public` value; since Scooter does not include arbitrary sum types, we instead encode this as two fields, a boolean `is_public` and a set of user `readers`.

LambdaChair. LambdaChair [58], a lightweight conference review system, is another Hails benchmark. It features Program Committee (PC) users, as well as non-PC users, both of whom can be paper authors. It also has a `root` principal that can edit any paper. The authors of LambdaChair evolved the LambdaChair application over time: first, they created authors and PC members, and then, through a migration, they added papers and permissions on those papers. However, they did so in ad-hoc way, by changing the policy and the model by hand. Scooter is able to express the LambdaChair migrations in thirty-eight lines of code, and Sidecar can quickly verify those migrations for safety.

Learn-by-Hacking. We port another Hails benchmark, Learn-by-Hacking [59], that lets users incorporate code into tutorials, blog posts, and more. The original authors evolved Learn-by-Hacking through five migrations (e.g., one adds tags (short categories) to associate with posts). Using our DSL alone, we are unable to express one migration using Scooter—that adds a database of existing tags—since this migration relies on querying and then dynamically creating objects. As we further discuss in Section 6.2, this migration can be implemented using the Scooter ORM.

UrFlow Calendar. We port UrFlow Calendar [7], an application that allows users to manage a calendar, from the UrFlow project [8] without any issues. Ur encodes policies very differently from Scooter—as a SQL-based eDSL—but Scooter is still able to express this benchmark's policies despite the difference.

Lifty Conference. We port Lifty Conference [60], another conference review system, from the Lifty project [40]. This benchmark is different from the others because Lifty is not an ORM; the benchmark operates on in-language values and types. The Lifty policies rely on a singleton which we translated into a database object. Scooter is able to express all policies from the Lifty benchmark.

5.2 Detecting Unsafe Migrations

Since none of the above case studies had unsafe schema or policy changes, we ensure that Sidecar can detect unsafe changes by implementing several unsafe schema and policy changes. First, our test suite contains multiple negative tests, including the unsafe Chitter application migrations described in Section 2. Second, we model two unsafe migrations from two applications: (1) a refactor of HotCRP's policy code that inadvertently granted unauthenticated users administrator rights [29, 30]; and (2) a policy change in the Hails Task management app that inadvertenly made projects readable to all users [49]. In all cases, Scooter sucesfully detects the unsafe migrations and generates counterexamples.

5.3 Sidecar Verification Speed

We evaluate the performance of Sidecar by timing the migrations from all of the case studies from Figure 5. The most expensive migration takes 88.8ms to verify; the fastest takes 10.3ms. Performing the safety checks on a given command, say `AddField`, takes 7.1–12.7ms.

Project	Framework	# Models	# Fields	# Migr	Migr LOC	Unique Policies	Migration Actions
BIBIFI	LWeb	46	215	11	183	4	37/37
Visit Days	Ruby on Rails	4	19	10	139	7	21/21
GitStar	Hails	3	8	1	11	7	6
LambdaChair	Hails	4	8	1	38	5	2/2
Learn-by-Hacking	Hails	3	13	5	63	7	22/23
Ur-Calendar	UrFlow	2	8	1	52	6	1/1
Lifty Conference	Lifty	6	26	1	175	10	1/1

Figure 5. A list of case studies, along with metrics. # Models is the number of models in the final policy; # Fields is the number of fields on all models in the final policy; # Migr is the number of migrations considered; Migr LOC is the total lines of code of migrations expressed in Scooter; Unique Policies indicates the count of unique policy functions that were ported to Scooter; Migration Actions indicates the ratio of migration actions that were expressible in Scooter.

5.4 ORM Performance Overhead

We measure the performance overhead of our ORM on two benchmarks—an end-to-end macro-benchmark and a micro-benchmark.

Macro-Benchmark. To understand the overhead our ORM imposes on real web applications, we port two BIBI controller benchmarks from LWeb [39] and measure the policy enforcement overhead on latency for each endpoint. Specifically, we port the `/announcements` route, which fetches announcements and schedules, and the `/profile` route, which fetches the logged-in user's profile. We use Scooter with the Rocket web framework (version 0.4) and Handlebar template system (version 1.1). To measure latency, we use the Apache benchmarking tool `ab`; we configure `ab` to make 10,000 requests with 16 concurrent connections. We find that the overhead on mean and tail latency, which we measure to be 4ms for both end points, is in the noise (< 0.1ms). This is not surprising: BIBIFI policies only allow field accesses and static principals—and thus checking whether a field can be accessed is answered by an equality check on already available data. Unlike Scooter, the overhead of enforcement in LWeb for these endpoints is 2.41–19.01%; this is likely because LWeb uses IFC, whereas Scooter only performs access-control checks at the database boundary.

Micro-Benchmark. Because BIBIFI's policies are not representative of the more complex policies supported by Scooter—policies that require database queries—we implement a micro-benchmark around the Chitter policy from Section 2. Specifically, we measure the ORM performance overhead by timing two different actions: (a) creating a Chitter post and (b) viewing a list of friends' posts. We do so in three configurations: (1) Unchecked: native database bindings with no policy checks; (2) Hand checked: manually written policy checks; and (3) Scooter checked: Scooter's ORM enforcement. We measure each action 10,000 times in each configuration and report the mean. As shown in Figure 6, our ORM is only slightly slower than manually inserted checks. In real web applications, these overheads are hidden by network latency and other application components.

Task	Create Post	View Friend Posts
Unchecked	0.313 ms	13.8 ms
Hand checked	0.334 ms	14.9 ms
Scooter checked	0.331 ms	15.2 ms

Figure 6. The time taken for two application level tasks, in the three configurations: unchecked, when manually checked, and when checked with our Rust ORM.

6 Discussion and Limitations

Like all languages and verifiers both Scooter and Sidecar have limitations. We discuss some of these next.

6.1 Expressiveness

Scooter tries to balance the need to express complex behavior with the need to be a reliable tool. Though we originally designed Scooter strictness checking to be decidable, we added language features—namely, set subtraction and cyclical models—which can be used to write policies that the SMT solver may not be able to solve. These features are useful for expressing certain data models and policies. For example set subtraction is necessary for expressing deny lists (in an open-world system where we cannot enumerate all principals). Though these features could cause Sidecar to time out (and thus require the developer to perform an unchecked migration), our verifier did not time out on any of the case studies or our tests.

There are many opportunities for additions to Scooter that increase expressivity without affecting decidability. For example, Scooter currently only support data types that can be used in policies. We envision extending Scooter with common datatypes (e.g., binary blobs, JSON objects, and geolocations) that cannot be referenced in polices—and thus Sidecar does not have to reason about—but are useful for many applications. Similarly, we envision extending Scooter

with anonymous records that would, for example, allow developers to express projections of database objects.

6.2 Data Migrations

Scooter does not support manipulating data in the database apart from the populating function in `AddField`. For example, developers can't create or edit objects in a migration script. Real-world migrations sometimes need to do this, though—and indeed we encountered this when porting Learn-by-Hacking to Scooter.

Verifying the safety of these operations is difficult because they can cause *indirect leakage*. For example, in a social media site where only a user's friends can see their email, a migration that creates a new friendship between two users also leaks their emails. In contrast to the kind of leakage Sidecar prevents, this leakage does not require sensitive data to flow to a permissive output. Instead, it modifies a policy-relevant field such that the permissions represented by an existing policy function are expanded.

Developers can work around this limitation by using the ORM to perform migrations at the application level, as a series of database queries. This ensures that all database accesses are protected by policies. In the rare case where developers need to elide policy enforcement, our ORM, in debug mode, also allows developers to temporarily turn-off enforcement.

6.3 Transactions

Scooter has no transactional semantics; any sequence of read, write, create, and delete actions must be valid at each step. This can create problems when multiple database mutations must happen together.

For example, in the LambdaChair case study, papers have multiple authors, and these authors (and only these authors) have permissions to add other authors. Without set fields, this would require three models: `Paper`, `User`, and `PaperAuthor` which represents the join table between them. When a user creates a paper, they need to create an instance of both `Paper` and `PaperAuthor`. If they create the `Paper` first, they would not be an author and thus could not add other authors. Conversely, if they try to create the join table entry first, there is no `Paper` to reference. In this specific case, we sidestep the problem by making authors a set field. Creating an instance with a set field is (to Scooter) a single action even though it maps to multiple database actions. In this way, set fields allow a specific type of transaction.

It is unclear how (or if) Scooter and Sidecar's approach to policy safety scales up to arbitrary database transactions, where multiple operations can occur atomically. We consider this future work.

6.4 Surprising Semantics

As described in Section 4, Sidecar tracks data equivalences during migrations. This allows policy-relevant fields to change representation—like `isAdmin` becoming `adminLevel`—and still pass verification. While this is a useful feature, it has subtle behavior that can produce surprising results.

The first surprise is that a sequence of migrations can be valid when it is in one file and invalid when it is split across two. Migration scripts are atomic, and thus Sidecar tracks equivalences across migration commands within a script file. But because writing to the database can invalidate equivalences, we do not track equivalences between scripts.

Which policies Sidecar considers equal, due to equivalences, may also be surprising. Consider the migration from `isAdmin` to `adminLevel` from Section 2, where admins are given `adminLevel` 2 and regular users are given `adminLevel` 0. Until `isAdmin` is removed from the model, Sidecar will track the relationship between the two fields. In doing so, Sidecar will deduce that:

```
u -> User::Find({isAdmin: true})
```

is equivalent to:

```
u -> User::Find({adminLevel: 2})
```

and, more surprisingly, also equivalent to:

```
u -> User::Find({adminLevel >= 1})
```

By tracking equivalence, Sidecar knows that—at the time of the migration—there are no users with an `adminLevel` 1 and that all users with `adminLevel` 2 were admins.

This could have unintended effects. Until `isAdmin` is removed, for example, we can use `{adminLevel >= 1}` instead of `{adminLevel: 2}` without Sidecar raising an alert—even if the semantic meanings of the two policies differ: one includes moderators, the other does not. Since equivalences let Sidecar succeed in many cases users expect it to and prevent them from relying on unchecked policy relaxations, we think this is a worthwhile trade-off. Of course, developers can turn equivalence tracking off; without them, Sidecar can still catch policy weakenings and leaks due to data flows but cannot incorporate knowledge from earlier migration steps.

7 Related Work

There is a long and rich history of work on the topic of security for database-backed application, and more broadly for general purpose applications. Broadly speaking our paper distinguishes itself from prior work in this space by offering a specific kind of enforcement that had not been previously explored: *static* verification of *data migrations* and *policy migrations* in database-backed applications. We see our effort as complimentary to the existing literature on secure database-backed applications. We now highlight the most closely related work in this area.

Dynamic Enforcement. Dynamic enforcement of security policies is a well-explored idea, including the seminal work on Execution Monitoring [46], hardware-level

information-flow tracking [52], language-based information-flow control [24, 51, 53], fine-grain discretionary access-control for databases [10, 35, 54, 66] and new programming models that incorporate dynamic enforcement [4, 65].

The dynamic approaches most closely related to our work are those developed for database-backed web frameworks (e.g., Jacqueline [64], Hails [20, 21], LWeb [39], and IFDB [47]). These frameworks allow the programmer to state policies separately from code, and the system automatically enforces the policy dynamically. While some of these systems enforce policies in a mandatory fashion (e.g., using IFC), Scooter only enforces access control policies at the ORM level, and thus our security guarantees do not extend to leaky application code. Since the ORM code is generated, though, Scooter could be extended to, say, generate ORM code that uses an IFC system for enforcement. Since our core focus is on verifying migrations, which none of these systems address, we see our work as largely tackling a complementary problem.

Static Enforcement. Another approach is to enforce policies statically, before the program runs. For example, there is a long line of work on static type systems that enforce fine-grained policies (e.g., information flow control and fine-grain access control) in various languages [1, 1, 3, 5, 6, 15, 23, 26, 32, 33, 33, 37, 40, 41, 44, 45, 50, 61]. The static approach most closely related to our work is UrFlow [8], a tool for static analysis of database-backed applications. UrFlow takes data policies specified as SQL queries, and statically ensures that the application code adheres to those policies. The Scooter ORM enforces policies dynamically and thus introduces runtime overheads that can be partially avoided with static checking. Conversely, neither UrFlow nor these other systems account for ways in which the applications (and, specifically, underlying database schema and policies) might change. Extending Scooter to systems like UrFlow that enforce policies statically is an interesting future direction.

Program Partitioning. Yet another approach to security enforcement is to use some kind of program partitioning, thus enforcing certain security properties by construction. For example, the Diesel [16] web framework for writing database-backed applications restricts each module of the application to use a database connection that can only access data in the corresponding database module. While this prevents certain kinds of cross-module leaks, it does not directly prevent secret leakage to an unprivileged user. Program partitioning can also be done automatically to enforce a stated policy, as is done in the Swift [9] system and the Jit/split [67] compiler. However, again, none of these approaches address data or policy migrations.

Schema Migration. There is a long history of work on schema migration. The most commonly explored topics include mechanisms for expressing schema mappings [31, 42], techniques for automatically inferring schema mappings [1, 2, 22, 36, 63], and support for automatically evolving and verifying queries and databases in the face of schema migration [11, 12, 62, 63]. None of this work addresses how security policies interact with migrations, which is the aim of our work. We, conversely, don't automatically update or verify the application code that uses the Scooter ORM.

Acknowledgments

We thank the reviewers, and our shepherd Adam Chlipala for his in-depth and insightful feedback on all aspects of the paper—and for finding a bug in one of our proofs. We thank Ranjit Jhala for his always insightful guidance. This work was supported in part by gifts from Cisco; by the NSF under Grant Numbers CCF-1918573, CCF-1955457, CNS-1514435, and CAREER CNS-2048262; and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Bogdan Alexe, Balder ten Cate, Phokion G. Kolaitis, and Wang-Chiew Tan. 2011. Designing and Refining Schema Mappings via Data Examples *(SIGMOD '11)*.

[2] Yuan An, Alex Borgida, Renée Miller, and John Mylopoulos. 2007. A Semantic Approach to Discovering Schema Mapping Expressions *(ICDE'07)*.

[3] O. Arden and A. C. Myers. 2016. A Calculus for Flow-Limited Authorization *(CSF'16)*.

[4] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-Agnostic Programs *(PLAS '13)*.

[5] Niklas Broberg, Bart van Delft, and David Sands. 2017. Paragon–Practical programming with information flow control. *Journal of Computer Security* 25 (2017).

[6] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing static and dynamic typing for information-flow control in Haskell *(ICFP'15)*.

[7] Adam Chlipala. [n.d.]. Ur/Flow Calendar Source Code. http://www.impredicative.com/ur/scdv/

[8] Adam Chlipala. 2010. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications *(OSDI'10)*.

[9] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure Web Applications via Automatic Partitioning *(SOSP '07)*.

[10] Brian J Corcoran, Nikhil Swamy, and Michael Hicks. 2009. Cross-tier, label-based security enforcement for web applications *(SIGMOD '09)*.

[11] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2013. Automating the database schema evolution process. *The VLDB Journal* 22 (2013).

[12] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. 2008. Graceful Database Schema Evolution: The PRISM Workbench *(VLDB'08)*.

[13] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver *(TACAS'08/ETAPS'08)*.

[14] Randall Degges. 2020. User Migration: The Definitive Guide. https://developer.okta.com/blog/2019/02/15/user-migration-the-definitive-guide.

[15] Dominique Devriese and Frank Piessens. 2011. Information flow enforcement in monadic libraries *(TLDI '11)*.

[16] Adrienne Felt, Matthew Finifter, Joel Weinberger, and David Wagner. 2011. Diesel: Applying Privilege Separation to Database Access *(ASIACCS '11)*.

[17] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.

[18] Ghost. 2018. Contributor should not be allowed to edit a post when not being a primary author. https://github.com/TryGhost/Ghost/issues/10238.

[19] Ghost. 2018. Fix access to the common article for multiple editors. https://github.com/TryGhost/Ghost/issues/10214.

[20] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. 2017. Hails: Protecting Data Privacy in Untrusted Web Applications. *Journal of Computer Security* 25 (2017).

[21] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. 2012. Hails: Protecting Data Privacy in Untrusted Web Applications *(OSDI'12)*.

[22] Georg Gottlob and Pierre Senellart. 2010. Schema Mapping Discovery from Data Instances. *J. ACM* 57, Article 6 (2010).

[23] Marco Guarnieri, Musard Balliu, Daniel Schoepe, David Basin, and Andrei Sabelfeld. 2019. Information-flow control for database-backed applications *(EuroS&P'19)*.

[24] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs *(SAC '14)*.

[25] Alicia Hope. 2020. Data Breach Index Site Leaks Over 23,000 Hacked Databases Exposing Over 13 Billion User Records. https://www.cpomagazine.com/cyber-security/data-breach-index-site-leaks-over-23000-hacked-databases-exposing-over-13-billion-user-records/.

[26] J. Hughes. 2000. Generalising monads to arrows. *Science of Computer Programming* 37 (2000).

[27] Troy Hunt. 2020. Have I Been Pwned: Check if your email has been compromised in a data breach. https://haveibeenpwned.com/.

[28] Peter Jonsson. 2003. *Automated Testing of Database Schema Migrations*. Master's thesis. KTH Royal Institute of Technology, Sweden.

[29] Eddie Kohler. 2014. Fix critical permissions error. https://github.com/kohler/hotcrp/commit/1e10f49687a9c6293f6365ca6e441e346cc2ab3d

[30] Eddie Kohler. 2014. Minor refactor. https://github.com/kohler/hotcrp/commit/6559c0cfef66c6dbe4b888971d7798cb7cca000e

[31] Phokion G. Kolaitis. 2005. Schema Mappings, Data Exchange, and Metadata Management *(PODS '05)*.

[32] Peng Li and Steve Zdancewic. 2006. Encoding Information Flow in Haskell *(CSFW '06)*.

[33] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. 2009. Fabric: a platform for secure distributed computation and storage *(SOSP'09)*.

[34] J. Mccarthy. 1962. Towards a Mathematical Science of Computation *(IFIP Congress)*.

[35] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. 2017. Qapla: Policy compliance for database-backed systems *(USENIX Security'17)*.

[36] Renée Miller, Laura Haas, and Mauricio Hernández. 2000. Schema Mapping as Query Discovery *(VLDB'00)*.

[37] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control *(POPL'99)*.

[38] Jaideep Nijjar and Tevfik Bultan. 2012. Unbounded Data Model Verification Using SMT Solvers *(ASE'12)*.

[39] James Parker, Niki Vazou, and Michael Hicks. 2019. LWeb: Information flow security for multi-tier web applications *(POPL'19)*.

[40] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid Information Flow Control. *PACMPL*, Article 105.

[41] François Pottier and Vincent Simonet. 2002. Information flow inference for ML *(POPL'02)*.

[42] Erhard Rahm and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *The VLDB Journal* 10, 4 (2001).

[43] John Renner. [n.d.]. Visit Day Source Code. https://github.com/PLSysSec/visit-day-rails/tree/530156cd95da1e791f49acbb99a01b23e714fd88

[44] Alejandro Russo. 2015. Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell *(ICFP'15)*.

[45] Alejandro Russo, Koen Claessen, and John Hughes. 2008. A Library for Light-weight Information-flow Security in Haskell *(Haskell'08)*.

[46] Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (Feb. 2000).

[47] David Schultz and Barbara Liskov. 2013. IFDB: decentralized information flow control for databases *(EuroSys'13)*.

[48] Alex Scroxton. 2020. Human error blamed in Welsh Covid-19 patient data leak. https://www.computerweekly.com/news/252492123/Human-error-blamed-in-Welsh-Covid-19-patient-data-leak.

[49] Amy Shen. 2013. Moved addUsers to policy module. https://github.com/a-shen/task/commit/9d9d806f7a2d6ece8f0191830cd6be67afdb1721

[50] E.G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F.B. Schneider. 2011. Logical attestation: an authorization architecture for trustworthy computing *(SOSP'11)*.

[51] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell *(Haskell'11)*.

[52] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking *(ASPLOS XI)*.

[53] Nikhil Swamy, Juan Chen, and Ravi Chugh. 2010. Enforcing Stateful Authorization and Information Flow Policies in Fine *(ESOP'10)*.

[54] Nikhil Swamy, Brian J Corcoran, and Michael Hicks. 2008. Fable: A language for enforcing user-defined security policies *(SP'08)*.

[55] Ryan Tate. 2010. Apple's Worst Security Breach: 114,000 iPad Owners Exposed. https://gawker.com/5559346/apples-worst-security-breach-114000-ipad-owners-exposed.

[56] BIBIFI Team. [n.d.]. BIBIFI Source Code. https://github.com/plum-umd/bibifi-code

[57] Hails Team. [n.d.]. GitStar Source Code. https://github.com/scslab/gitstar/tree/589663b54d26468a9407be3d9d5ffa4dfa4962a8

[58] Hails Team. [n.d.]. LambdaChair Source Code. https://github.com/deian/lambdachair/tree/660351f86d1dd603e551237d58d97d419e05bbec

[59] Hails Team. [n.d.]. LearnByHacking Source Code. https://github.com/deian/lbh/tree/84994a693ceec8d171adfe3d3f08528cd964c73a

[60] Lifty Team. [n.d.]. Lifty Conference Source Code. https://github.com/nadia-polikarpova/synquid/blob/4cafb45df74f8c45739994466be874918de38915/test/conference/Conference.sq

[61] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. 2018. MAC: a verified static information-flow control library. *Journal of logical and algebraic methods in programming* 95 (2018).

[62] Yuepeng Wang, Isil Dillig, Shuvendu K Lahiri, and William R Cook. 2017. Verifying equivalence of database-driven applications *(POPL'17)*.

[63] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration Using Datalog Program Synthesis *(VLDB'20)*.

[64] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-backed Applications *(PLDI'16)*.

[65] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A language for automatically enforcing privacy policies *(POPL'12)*.

[66] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2009. Improving application security with data flow assertions *(SOSP'09)*.

[67] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2002. Secure Program Partitioning. *ACM Trans. Comput. Syst.* 20, 3 (2002).

A Eliminating Sets from Scooter

1. Eliminate Set Fields. Prior to any verification, Sidecar removes set fields by rewriting them to join tables and rewriting field accesses:

```
foo.set ⇒
  FooSet::Find({from: foo.id}).map(j → j.to)
```

and queries :

```
Foo::Find(..., set ∋ e) ⇒
  Foo::Find(...).flat_map(f →
  FooSet::Find({from: f.id, to: e}).map(j → f))
```

Using these rewritten policies, we define leakage:

Definition A.1 (Leakage)**.** We define the leakage L as a predicate over two policies c_1, c_2 so that $L(c_1, c_2)$ is true if policy c_2 permits principals that c_1 does not. Formally,

$$L(c_1, c_2) = \exists D : Set(Record), p : Principal, r \in D.$$
$$D \vdash p \in c_2(r) \land \neg(p \in c_1(r))$$

Here, D represents the database, which contains all values that are returned by `Find` and `ById` queries.

2. Eliminate Sets. We now eliminate all higher-order operations, `Find` queries, and set-typed expressions. We do this with Dist a meta-function that distributes the $\in$ operator across all Scooter expressions. (Dist is a big-step application of $\rightsquigarrow$ from Section 4.) We then show that the transformed expressions are semantically equivalent to the original expressions—and, in the end, that all set-typed expressions are eliminated.

Definition A.2 (Dist)**.** Let Dist be a function over expressions of the form $x \in e$ defined as:

$$\textsc{Dist}[x \in e_1 + e_2] = \textsc{Dist}[x \in e_1] \lor \textsc{Dist}[x \in e_2] \quad (1)$$

$$\textsc{Dist}[x \in e_1 - e_2] = \textsc{Dist}[x \in e_1] \land \neg\textsc{Dist}[x \in e_2] \quad (2)$$

$$\textsc{Dist}[x \in (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3)] = \texttt{if } e_1 \texttt{ then } \textsc{Dist}[x \in e_2] \texttt{ else } \textsc{Dist}[x \in e_3] \quad (3)$$

$$\textsc{Dist}[x \in (\texttt{match } e_1 \texttt{ as } y \texttt{ in } e_2 \texttt{ else } e_3)] = \texttt{match } e_1 \texttt{ as } y \texttt{ in } \textsc{Dist}[x \in e_2] \texttt{ else } \textsc{Dist}[x \in e_3] \quad (4)$$

$$\textsc{Dist}[x \in e_1.\texttt{map}(a \rightarrow e_2)] = \exists v.\ \textsc{Dist}[v \in e_1] \land (x = e_2[v/a]) \quad (5)$$

$$\textsc{Dist}[x \in e_1.\texttt{flat\_map}(a \rightarrow e_2)] = \exists v.\ \textsc{Dist}[v \in e_1] \land \textsc{Dist}[x \in e_2[v/a]] \quad (6)$$

$$\textsc{Dist}[x \in [e_1, ..., e_n]] = (x = e_1) \lor ... \lor (x = e_n) \quad (7)$$

$$\textsc{Dist}[x \in \mathsf{T} :: \texttt{Find}(\{k_1\ cmp_1\ e_1, ..., k_n\ cmp_n\ e_n\})] = (p.k_1\ cmp_1\ e_1) \land ... \land (p.k_n\ cmp_n\ e_n) \quad (8)$$

Note that Dist is (and only needs to be) defined for all expressions of type $\texttt{Set}(\tau)$. If e were of some other type, then $x \in e$ would be malformed.

Theorem A.3 (Dist is semantics preserving)**.** $\textsc{Dist}[p \in e] \iff (p \in e)$

Proof. Cases (1)-(4) and (7) are trivially equivalent by basic set properties. Cases (5) and (6) rely not only on basic set semantics but the semantics of the `map` operator. The `map` operator applies some function f to every element of an input set to construct a new output set. For $x \in e_1.\texttt{map}(f)$ to be true, there must be some element $v \in e_1$ such that $f(v) = x$. Conceptually, this means we only have to witness a single element from the input set, such that when it is mapped, we get x. This reasoning extends to `flat_map` except we require that x is in the output set rather than equal to the output value. Case (8) eliminates the database lookup. Instead, we rely on an invariant that all instances (used in policy expressions) are present in the database. Any instance x is, by construction, present in the database. So, for $x \in \mathsf{T} :: \texttt{Find}(...)$ to be true, we can simply check if x would be returned by the find query, i.e., if x meets the query's criteria. □

Corollary A.4. *From Theorem A.3, we can define a new leakage model L', where $L' \iff L$, as:*

$$L'(c_1, c_2) = \exists D : Set(Record), p : Principal, r \in D.$$
$$D \vdash \textit{Dist}[p \in c_2(r)] \land \neg \textit{Dist}[p \in c_1(r)]$$

Next, we show that Dist-transformed expressions do not contain any set-typed subexpressions.

Theorem A.5. *Dist elminates all expressions of type $Set(\tau)$. That is, $\forall p,\ e.\ \nexists e_s \in subexprs(\textit{Dist}[p \in e]).\ \Gamma \vdash e_s : Set(\tau)$, where $subexprs(e)$ is the set of all subexpressions in e.*

Proof. By construction, expressions that are not of type $Set(\tau)$ cannot contain subexpressions of type $Set(\tau)$. This is true by induction over the grammar of Scooter and amounts to the property that there is no expression that reduces a set to a base value. Then, by induction over the cases of Dist:

1. Given e is of type $Set(b)$, all recursive applications of Dist are also over expressions of type $Set(b)$. This is trivial in all cases except for `map` and `flat_map` which use a new target v in the recursive call over e_1. By the typing rules for lambdas, parameters must be base types, thus v is also a base type.
2. All expressions not transformed by Dist (e.g., `if` conditions and `match` values) are, by construction, of base type. For the `T::Find` case, we eliminate set fields, so all fields (and corresponding expressions) are of base type.

Because Dist recurses over a finite AST, it is guaranteed to terminate and eliminate all set-typed expressions. □

B Scooter Grammar and Typesystem

(base type) $b ::= \texttt{String} \mid \texttt{DateTime} \mid \texttt{I64} \mid \texttt{F64} \mid \texttt{Bool} \mid \texttt{Id}(\mathit{model}) \mid \texttt{Option}(b)$
(type) $\tau ::= b \mid \texttt{Set}(b)$
(policy value) $\mathit{pol} ::= \texttt{public} \mid \texttt{none} \mid \mathit{func}$
(field declarations) $\mathit{field} ::= \mathit{ident} : \tau\ \{\texttt{read:}\mathit{pol}, \texttt{write:}\mathit{pol},\}$
(model) $\mathit{model} ::= \texttt{@principal?}\ \mathit{ident}\ \{\texttt{create:}\ \mathit{pol}, \texttt{delete:}\ \mathit{pol},\ < \mathit{field}, * > \}$
(static principal) $\mathit{sp} ::= \texttt{@static} - \texttt{principal}\ \mathit{ident}$
(policy file) $\mathit{pf} ::= < \mathit{sp}* > < \mathit{model}* >$

Figure 7. The syntax of policies in Scooter.

$\mathit{cmd} ::= \texttt{CreateModel}(\mathit{model}) \mid \texttt{DeleteModel}(\mathit{ident}) \mid \texttt{AddPrincipal}(\mathit{model}) \mid \texttt{AddStaticPrincipal}(\mathit{ident}) \mid \mathit{ident}::\mathit{mc}$
$\mathit{mc} ::= \texttt{AddField}(\mathit{field}) \mid \texttt{RemoveField}(\mathit{ident}) \mid \texttt{RenameField}(\mathit{ident},\ \mathit{ident})$
$(\texttt{Update}|\texttt{Weaken})\texttt{Policy}(\{\texttt{create:}\mathit{pol},\ \texttt{delete:}\mathit{pol}\}) \mid (\texttt{Update}|\texttt{Weaken})(\texttt{Create}|\texttt{Delete})\texttt{Policy}(\mathit{pol})$
$(\texttt{Update}|\texttt{Weaken})\texttt{FieldPolicy}(\{\texttt{read:}\mathit{pol},\ \texttt{write:}\mathit{pol}\}) \mid (\texttt{Update}|\texttt{Weaken})\texttt{Field}(\texttt{Read}|\texttt{Write})\texttt{Policy}(\mathit{ident}, \mathit{pol})$

Figure 8. The syntax of migrations in Scooter.

(base type) $b ::= \texttt{String} \mid \texttt{DateTime} \mid \texttt{I64} \mid \texttt{F64} \mid \texttt{Bool} \mid \texttt{Id}(\mathit{model}) \mid \texttt{Option}(b) \mid \texttt{Principal} \mid \texttt{Object}(\mathit{model})$
(type) $\tau ::= b \mid \texttt{Set}(b) \mid \tau \to \tau$

$$\frac{\tau \text{ is } \texttt{I64} \text{ or } \texttt{F64}}{\mathit{numpcmp} : \tau \to (\tau \to \texttt{Bool})} \qquad \frac{\tau \text{ is } \texttt{I64} \text{ or } \texttt{F64}}{\mathit{numop} : \tau \to (\tau \to \tau)} \qquad \frac{}{\mathit{gencmp} : b \to (b \to \texttt{Bool})}$$

$$\frac{\tau \text{ is } \texttt{I64} \text{ or } \texttt{F64} \text{ or } \texttt{Set}(\tau)}{\mathit{op} : \tau \to (\tau \to \tau)} \qquad \frac{\mathit{binop} : \tau_1 \to (\tau_2 \to \tau_3) \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ \mathit{binop}\ e_2 : \tau_3} \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}} \qquad \frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma \vdash !e : \texttt{Bool}} \qquad \frac{\Gamma \vdash e_{cond} : \texttt{Bool} \qquad \Gamma \vdash e_{true}, e_{false} : \tau}{\Gamma \vdash (\texttt{if}\ e_{cond}\ \texttt{then}\ e_{true}\ \texttt{else}\ e_{false}) : \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Some}(e) : \texttt{Option}(\tau)}$$

$$\frac{}{\Gamma \vdash \texttt{None} : \texttt{Option}(\tau)} \qquad \frac{\Gamma \vdash e_t : \texttt{Option}(\tau_1) \qquad \Gamma, v : \tau_1 \vdash e_{some} : \tau_2 \qquad \Gamma \vdash e_{none} : \tau_2}{\Gamma \vdash (\texttt{match}\ e_t\ \texttt{as}\ v\ \texttt{in}\ e_{some}\ \texttt{else}\ e_{none}) : \tau_2} \qquad \frac{\Gamma, v : b \vdash e : \tau}{\Gamma \vdash v \to e : b \to \tau}$$

$$\frac{\Gamma \vdash e_0, e_1, .., e_n : \tau}{[e_0, e_1, .., e_n] : \texttt{Set}(\tau)} \qquad \frac{\Gamma \vdash e : \texttt{Set}(\tau_1) \quad \Gamma \vdash f : \tau_1 \to \tau_2}{\Gamma \vdash e.\texttt{map}(f) : \texttt{Set}(\tau_2)} \qquad \frac{\Gamma \vdash e : \texttt{Set}(\tau_1) \qquad \Gamma \vdash f : \tau_1 \to \texttt{Set}(\tau_2)}{\Gamma \vdash e.\texttt{flat\_map}(f) : \texttt{Set}(\tau_2)}$$

$$\frac{\Gamma \vdash e : \texttt{Object}(M) \qquad M \text{ has field } f\text{: } \tau}{\Gamma \vdash e.f : \tau} \qquad \frac{\Gamma \vdash e : \mathit{Id}(M)}{\Gamma \vdash M\text{::}\ \texttt{ById}(e) : \mathit{Object}(M)} \qquad \frac{}{:: b \to (b \to \texttt{Bool})}$$

$$\frac{}{\ni : \texttt{Set}(\tau) \to (\tau \to \texttt{Bool})} \qquad \frac{\forall i.\ M \text{ has a field } f_i : \tau_{if} \qquad \Gamma \vdash e_i : \tau_{ie} \qquad \mathit{fop}_i : \tau_{if} \to (\tau_{ie} \to \mathit{Bool})}{\Gamma \vdash \texttt{M} :: \texttt{Find}(\{f_1\ \mathit{fop}_1\ e_1, ..., f_n\ \mathit{fop}_n\ e_n\}) : \mathit{Set}(\mathit{Object}(M))}$$

$$\frac{\Gamma \vdash e : \tau_s \qquad \tau_s <: \tau}{\Gamma \vdash e : \tau} \qquad \frac{}{\texttt{I64} <: \texttt{F64}} \qquad \frac{\text{M is a principal model}}{\texttt{Id}(M) <: \texttt{Principal}} \qquad \frac{\tau_s <: \tau}{\texttt{Set}(\tau_s) <: \texttt{Set}(\tau)}$$

Figure 9. Typing rules for Scooter policies