A Framework for Specifying and Formally Verifying Application Security Policies

by

Christopher Shao

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author .....
Department of Electrical Engineering and Computer Science
May 24, 2019

Certified by .....
Adam Chlipala
Associate Professor of Computer Science
Thesis Supervisor
May 24, 2019

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee
May 24, 2019

A Framework for Specifying and Formally Verifying Application Security Policies

by

Christopher Shao

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

One challenge of building software applications that handle sensitive information is ensuring that they meet certain security and privacy policies, which guide how the application should be written in order to satisfy particular security properties. Common examples of security policies include information-flow control and access control. In complex applications, which can be composed of many stateful components, it is hard to reason about all possible interactions and check that a policy is satisfied in every case. In this thesis, we present work on a new static-analysis framework in the Coq proof assistant to verify that implementations of applications meet their specified security policies, using proofs of indistinguishability of labeled transition systems. The primary goal of our framework is to be applicable to a wide variety of applications and policies, moreso than existing analysis tools. In addition to a formalization of applications and policies, we discuss some theorems to reduce manual proof effort and enable modular development. Finally, we apply our framework to some simple examples.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor of Computer Science

Acknowledgments

I would like to thank Adam Chlipala, my advisor, for his invaluable guidance and advice during my research career. I would like to thank John Grosen, with whom I worked on this project, for all of our interesting and productive discussions. I would like to thank my family for always supporting me and giving me the opportunities I have today. Finally, I would like to thank my friends for all of the fun times we shared together.

Contents

List of Figures

Chapter 1

Introduction

Computer applications that handle sensitive data are typically designed with some security policies in mind, to enforce certain properties about how the application manipulates data. Common security policies include: access control, restricting classified data to certain parties with sufficient privileges; channel control, ensuring that information follows certain paths through an application; and integrity control, preventing data from being modified by unauthorized parties. If an application does not properly meet its security policy, it is possible that malicious parties could access or modify sensitive data in ways unintended by the developer. So it is important for a developer to check that the application's implementation does indeed follow the policies in all cases. However, in today's complex web or mobile applications, information can be manipulated and transferred in many different ways, making it hard manually to keep track of all ways information is managed.

One solution to this problem is to do static checking on an application's source code to show that it meets a security policy. However, many static-analysis tools for checking security policies only work for specific security guarantees or specific types of applications (see Chapter 6). These tools are often not general enough to support policies that are application-specific and more ad-hoc.

This thesis presents a different static analysis framework for verifying security policies, one that we believe can feasibly verify both general and application-specific policies for a wide variety of applications.

1.1 Coq

We have developed our framework in the Coq proof assistant [1]. Coq is a computerized proof assistant that provides a language Gallina to write executable programs and mathematical definitions, and a proof tactic language Ltac to write proofs and proof scripts. Developers using Coq can prove theorems about the behavior of their programs, by manually building proofs whose individual steps are checked by Coq. Gallina prioritizes having a rigorous underlying logic and type system, and as a result, native Gallina programs lack some features of conventional programming languages, such as non-terminating functions and exceptions. So in some applications of Coq, such as Kami [2] and FSCQ [3], that reason about more conventional programming-language features, developers first write programs in some object language embedded in Coq, define the language's semantics, and prove theorems about how these programs would run according to their semantics, all in Coq. In a similar way, we have developed a language embedded in Coq for writing applications and policies as well as theorems and automated tactics for proving that an application meets a policy.

1.2 Collaborators and Contributions

The work on this project was done by Christopher Shao, the author of this thesis, and John Grosen, with additional design input from Clément Pit-Claudel. John primarily contributed the definitions of both the syntax and semantics of the language in Sections 3.1 and 3.2, as well as the definitions of bisimulation and relational logic in Sections 4.1 and 4.2. The author's work mainly builds upon the definitions contributed by John, through proof work in Section 4.3 and in Chapter 5. The commentary in Section 3.3 is also drawn from the author's experience working with the framework.

Chapter 2

Overview

In this section, we give a high-level overview of our approach, before describing the Coq implementation in detail in the next section.

In our framework, we model applications as collections of modules that can interact with each other. Each module has its own internal private state and implementations of methods that can compute with or modify this state. Depending on the application, modules can also make calls to other methods of certain other modules as part of their implementation. The exact modules that a specific module has the capability to call are specified by the application structure. In order to model realistic applications, which may expose only certain modules to the outside world or make calls to external services whose implementations are not available, the modules in an application are designated as either public, internal, or external. **Public** modules are modules whose methods are intended to be called by users of the application. **Internal** modules have methods that can be called as intermediate steps during computation but are not exposed to users. **External** modules have methods without available implementations, and each external module is treated essentially as a black box that returns some value when one of its methods is called. Examples of external modules could be third party libraries or an output-logging device that exposes some interface, like a printer. Of note is that all the methods of public and internal modules must have implementations provided.

To compute with an application, a user would call a method of a public mod-

ule, say A, and A computes based on its implementation and private state. During computation, A may make a method call to some other module B, at which point A waits for that method call to finish and return some value, and then continues. Since each module has state that can be changed, calling the same method on a module at different times may yield different results and computation, depending on the internal state. So a user's interaction with an application consists of a sequence of public method calls one after the other, updating the states of the application modules in the process.

Instead of formulating a conceptually different structure to specify security policies, we propose to model them in a way that mirrors applications. In our framework, a policy is also modeled as a collection of modules that interact with each other, except some modules may be left unimplemented.

From a single policy, we can derive a whole class of applications by "filling in" the implementations for the unimplemented modules and methods in the policy. We call any one of these applications an **instantiation** of the policy. By manually inspecting the structure and implementations in a policy, a developer can derive security properties of any instantiation of the policy, without having to reason about the functionality of the modules left unimplemented in the policy.

Our model for policies is motivated by the observation that many real applications have both components dealing with security and components that serve core functionality, the latter of which are mostly irrelevant to the overall security of the application. For example, many websites or services require users to authenticate themselves before accessing user-specific information, enforcing the guarantee that "information specific to a user is only given once that user has been authenticated". Once authenticated, the rest of the application can freely serve user-specific information, knowing that the user has been authenticated and that the aforementioned guarantee will be enforced regardless of what else the application does. So to enforce security in this case, it is only necessary to verify the authentication components work as intended.

Thus in our model, it is intended that policies contain implementations for only the

modules relevant to security. Removing information irrelevant to security makes a policy simpler for a human to audit than a fully implemented application. Furthermore, a wider variety of applications with different implementations can be instantiations of the same policy, and thus satisfy the same security guarantee.

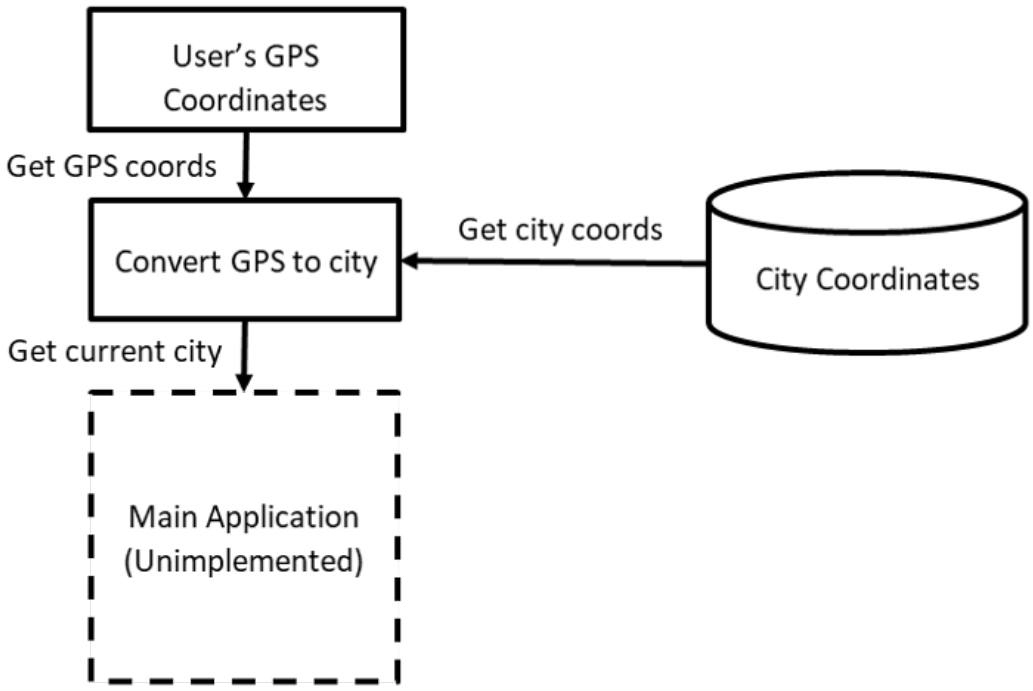

Figure 2-1: A security policy for an application that requires the user's city.

Figure 2-1 shows a possible policy for an application that changes its behavior based on the user's current city, as derived from the user's GPS coordinates. Here, the modules are represented by shapes, and an arrow from a module A to a module B means that B has the capability to call a method of A and receive a return value back. A potentially important security guarantee for such an application would be that the main user interface does not directly see the user's exact GPS coordinates, but instead it only sees the user's current city. With this guarantee, it is impossible for the main user interface to leak the user's exact coordinates, which prevents an adversary from, for example, stalking a user's exact position using the application.

It is evident that any instantiation of the policy in Figure 2-1 satisfies the desired security guarantee, since the main user interface never obtains exact GPS coordinates and only indirectly gains information about the coordinates through the module returning the current city. Here, we assume that the modules that return the user's GPS coordinates, return coordinates of cities, and convert a user's GPS coordinates to a city are implemented correctly (e.g. the converting module does not just return the user's GPS coordinates). Their implementations are included as part of the policy, since their implementations are relevant to the desired security guarantee while the main user interface is not. A developer can and should check their implementations when verifying that this policy does enforce the desired security guarantee.

For a given policy P that enforces some security guarantee, not only do instantiations of P meet the guarantee, applications that are **indistinguishable** from an instantiation should also meet the guarantee. Here, applications only need to be indistinguishable to an outside observer of the application (like a user) who does not see the internal structure of the application. More precisely, we assume that an observer of an application can only see the public and external modules of an application and can only interact with the application by making method calls on public modules. Then, we consider two applications to be indistinguishable if they have the same public and external module signatures, and for every sequence of public method calls, the applications produce the same **trace**. For our purposes, traces consist of not only return values of methods but also the results of calls to external modules made during the computation. We include the results of calls to external modules in the trace since it is possible that an application could make a call to an external service running elsewhere, and an observer could see the outgoing traffic. This definition is also known as **trace equivalence**. With this definition of trace, we assume that an application could leak information about its computation only through the return values or calls to external modules, and we concern ourselves with security guarantees that restrict information leaving an application through these methods. We do not model other observable side effects, and so this framework cannot guarantee an application is safe from side-channel attacks like timing attacks.

With our definition of indistinguishability, an observer of two indistinguishable applications cannot distinguish them by differences in public-facing module signatures, nor by observable differences in how the applications run on public-facing methods. Then, an application indistinguishable from an instantiation of the policy, even with a different internal module structure, satisfies the same security guarantee, at least from the perspective of a user that only interacts with the application through public modules. So we say that an application meets a security policy if it is indistinguishable from some instantiation of the policy.

Modeling both applications and policies as collections of interacting modules has several advantages. First, it enables us to specify and verify a wide variety of applications and policies, particularly policies that are application-specific and not easily encapsulated by general categories like information-flow control. Second, if policies are designed to make some security guarantees intuitively obvious, it suffices to prove indistinguishability to prove security. For example, an intuitive way to restrict information flow between two modules is by not providing a path between them in the policy. Finally, by framing the problem of satisfying a policy as one of showing indistinguishability of two transition systems, we can approach the problem by applying well-studied techniques in formal methods for showing equivalence of programs.

Chapter 3

Coq Formalization

In this section, we discuss our implementation language for applications and policies, as described in the overview.

Preliminary Notation

The next several chapters will feature many excerpts of our Coq code, some of which are abridged or simplified for clarity. In addition to notation for expressions, methods, and modules (explained later), we use the following notations:

- `StringMap` is an implementation of finite maps keyed by strings, following the finite map interface in Coq's standard library.
- `m $? k` denotes `StringMap.find k m`.
- `m $+ (k, v)` denotes `StringMap.add k v m`.

3.1 Syntax

3.1.1 Expressions

First, the values in our language take on one of a limited set of types, given by `type` below, that correspond naturally to several Coq types. `interp_type` formalizes this correspondence.

```
Inductive type : Type :=
| TyUnit
| TyNat
| TyString
| TyBool
| TyTuple (ty1 ty2 : type)
| TyList (ty : type).

Fixpoint interp_type (t : type) : Type :=
  match t with
  | TyUnit ⇒ unit
  | TyNat ⇒ nat
  | TyString ⇒ string
  | TyBool ⇒ bool
  | TyTuple t1 t2 ⇒ interp_type t1 * interp_type t2
  | TyList t' ⇒ list (interp_type t')
  end.
```

Next, an expression describes a computation within a single module, given by `expr stateTy ty` below. An expression can be thought of as the implementation of a method of some module, and so the expression can manipulate the state of its module as well as return values. `expr` is parametrized by two types: `stateTy`, the type of the private state of the expression's enclosing module; and `ty`, the type to which the expression evaluates.

Note that expressions (and as a result, methods and modules later) are defined with PHOAS [4] terms, meaning that they are also parametrized by a function `var : type → Type` that determines the types of variables.

```
Inductive expr (var : type → Type) (stateTy : type) : type → Type :=
| Var : ∀ ty, var ty → expr var stateTy ty
| Const : ∀ ty, interp_type ty → expr var stateTy ty
| Apply : ∀ ty1 ty2,
    (interp_type ty1 → interp_type ty2) →
    expr var stateTy ty1 →
    expr var stateTy ty2
| Build : ∀ ty1 ty2,
    expr var stateTy ty1 →
    expr var stateTy ty2 →
    expr var stateTy (ty1 * ty2)%ptype
| Nil : ∀ ty, expr var stateTy (TyList ty)
| Cons : ∀ ty,
    expr var stateTy ty →
    expr var stateTy (TyList ty) →
    expr var stateTy (TyList ty)
| If : ∀ ty,
    expr var stateTy TyBool →
    expr var stateTy ty →
    expr var stateTy ty →
    expr var stateTy ty
| While : expr var stateTy TyBool →
    expr var stateTy TyUnit →
    expr var stateTy TyUnit
| MCall : ∀ argTy retTy (m meth : string),
    expr var stateTy argTy →
    expr var stateTy retTy
| LetIn : ∀ ty1 ty2,
    expr var stateTy ty1 →
    (var ty1 → expr var stateTy ty2) →
    expr var stateTy ty2
| GetState : expr var stateTy stateTy
| SetState : expr var stateTy stateTy → expr var stateTy TyUnit.
```

Here, we explain in more detail what every constructor means.

- `Const` wraps a Coq constant into an expression, and `Apply` applies a *pure* Coq function to a single expression. These constructs allow Coq terms to be injected directly into expressions, simplifying the process of writing expressions. By suitably packaging and unpackaging multiple arguments as a single tuple, `Apply`

can also be used to apply functions with multiple arguments.

- `Build` is the constructor for tuples. The projection functions `Fst` and `Snd` can be defined in terms of `Apply`, and so they need not be primitive in the language. `Build` must be primitive, as it is needed to package multiple arguments for `Apply`.

- `Nil` and `Cons` are the constructors for lists.

- `If` and `While` are control-flow constructs.

- `MCall` represents a method call to another module (either the same as or different from the enclosing module), which takes one argument and returns one value. `argTy` and `retTy` are the types of the argument and return value, respectively. Once again, by packaging and unpackaging arguments as a single tuple, multiple arguments can be passed to `MCall`.

  Foreshadowing the discussion of our module-structure formalization, modules and methods are referred to simply by Coq `string`s. The two additional string arguments, `m` and `meth`, of `MCall` are the names of the called module and the called method, respectively.

  Having `argTy`, `retTy`, `m`, and `meth` as arguments to every `MCall` raises some questions about the consistency of `MCall`s over multiple modules. How do we know that `m` and `meth` refer to a method with signature given by `argTy` and `retTy`? How do we know that all calls to the method `meth` have the same signature? How do we know that `m` and `meth` refer to a method that exists at all? Expressions describe a single module's computation, and to answer these questions, we must know what other modules exist outside of the expression's enclosing module. Therefore, we can only check the consistency of `MCall`s in the context of an entire application. This consistency check is captured by `valid_module` (see Section 3.1.4).

- `LetIn` is a let binding, which serves to sequence multiple expressions together.

- Finally, `GetState` and `SetState` are used to read and write the private state of the enclosing module.

Expression Notation

Using Coq's built-in notation extension capabilities, we use custom notation for `expr` to make writing expressions more natural.

```
Notation "# v" := (Var v) (at level 0).
Notation "$ e" := (Const e) (at level 0).
Infix "!" := (Apply) (at level 0).

Notation "'plet' name := expr 'in' cont" :=
  (LetIn expr (fun name ⇒ cont))
    (at level 12, right associativity, name at level 0).
Notation "'call' modname # meth ( arg )" :=
  (MCall modname meth arg)
    (at level 12, modname at level 0, meth at level 0).
Notation "'read'" := (GetState) (at level 12).
Notation "'write' e" := (SetState e) (at level 12).
Notation "'if_' ( c ) 'then_' { e1 } 'else_' { e2 }" :=
  (If c e1 e2) (at level 12).
```

For example, the two expressions below are identical.

```
Definition test_expr :=
  LetIn GetState
    (fun a ⇒
     LetIn (Build (Const 3) (Const 4))
       (fun b ⇒
        If (Apply (fun x ⇒ Nat.eqb (fst x) (snd x)) ! (Var b))
          (MCall "mod" "meth" (Var a))
          (SetState Nil))).

Definition test_expr_notation :=
  plet a := (read) in
  plet b := (Build $3 $4) in
  if_ ( (fun x ⇒ Nat.eqb (fst x) (snd x)) ! (#b) )
    then_ { call "mod"#"meth" ( #a ) }
    else_ { write Nil }.
```

3.1.2 Methods

A method takes one argument (again, it is possible to emulate multiple arguments with tuples) and returns one value. Its implementation is represented by an `expr` that has one argument variable with type given by the signature, where the `expr` has return type also given by the signature.

```
Record methodsig : Type :=
  { MethArg : type;
    MethRet : type; }.

Definition method (stateTy : type) (methtype : methodsig) : Type :=
  var (MethArg methtype) → (* method argument *)
  expr stateTy (MethRet methtype).
```

Every module has some number of methods, each with its own name as a Coq string, and all the methods for a module can be encapsulated with an instance of `methods` below. The core idea of our framework is that policies and applications are written in the same language, with the added ability for policies to leave some methods or modules unimplemented. Thus `methods`, as well as the definition of modules later, can account for methods with and without implementations and can be used to specify both policies and applications. They are parametrized by an extra Boolean, indicating whether they are a part of an application (`true`) or a policy (`false`).

```
Inductive methods (stateTy : type) : bool → Type :=
| MethodsNil : ∀ c, methods stateTy c
| MethodsNoImpl : ∀ (methtype : methodsig) (name : string),
    list string →
    methods stateTy false →
    methods stateTy false
| MethodsImpl : ∀ c methtype (name : string),
    method stateTy methtype →
    methods stateTy c →
    methods stateTy c.
```

`MethodsImpl` specifies a method with its name and implementation. `MethodsNoImpl` omits the implementation of a method, requiring only its signature, name, and a

whitelist of names of modules that the method would be allowed to call. It is important that the signature of an unimplemented method is still documented, so that calls to this method from other modules can be type-checked. The list of allowed modules is also needed to specify the capabilities of an unimplemented method in a policy, allowing the writer of a policy more fine-grained control over unimplemented methods and their security properties. Note that using `MethodsNoImpl` forces the type to be `methods stateTy false`, meaning that it can only be used in a policy.

3.1.3 Modules

To organize modules, our framework uses a hierarchical module system not unlike that of OCaml. Base modules implement methods and have private state. These base modules can be composed together through let bindings, in which one module is declared in scope of a second module. The key feature of this definition is that the capabilities of modules are enforced by *scoping rules*. In particular, module A can only call a method of module B if B was previously declared in scope of A.

```
Inductive module (var : type → Type) : bool → Type :=
| ModuleLet : ∀ c,
    string →
    module var c →
    module var c →
    module var c
| ModuleLetRec : ∀ c,
    StringMap.t (module var c) →
    module var c →
    module var c
| ModuleAnyState :
    (∀  stateTy : type, methods var stateTy false) →
    module var false
| ModuleState : ∀ c stateTy,
    interp_type stateTy → (* initial state value *)
    methods var stateTy c →
    module var c
```

- `ModuleAnyState` and `ModuleState` are the base modules. `ModuleState` requires an initial value for its state, as well as its methods. `ModuleAnyState` does not require a fixed state type, and it exists so that policies can also leave a module's state unimplemented. Similar to `MethodsNoImpl`, `ModuleAnyState` forces the type to be `module var false`, since only policies can omit a module's state.

- `ModuleLet` and `ModuleLetRec` are the combinators. In `ModuleLet name m1 m2`, `m1` is bound to the name `name`, and `m2` can then call `m1` by using `name` as the module name. Of note is that `m1` cannot call a method of `m2`, as `m2` is not in scope of `m1`, and it also cannot call itself. In `ModuleLetRec submods m`, `submods` maps names to modules, binding possibly any number of modules to names. The modules in `submods` are all considered in scope of each other, and thus any module in `submods` can call a method of another module in `submods` (even itself). This allows applications to have modules that are self-recursive or mutually recursive.

  Notably, the syntax allows for a name to be reused when binding modules. For example, in `ModuleLet name m1 (ModuleLet name m2 m3)`, both `m1` and `m2` have been bound to `name` from the perspective of `m3`. While other languages allow shadowing in cases like this, we choose to disallow shadowing to avoid confusion when reading application or policy definitions. We check that there are no name collisions with `valid_module` (see Section 3.1.4).

We clarify some of the scoping rules of `module` with the following example.

```
Definition scoping_example :=
  ModuleLet name1 (ModuleLet name2 m1 m2) (ModuleLet name3 m3 m4).
```

Here, `m1`, `m2`, etc. are basic modules. The leftmost `ModuleLet` binds the name `name1` to `ModuleLet name2 m1 m2`, so `name1` is bound to some module in the scopes of `m3` and `m4`. Then, `m3` and `m4` can call the module named `name1`.

However, `m3` and `m4` can only directly call the methods of `m2` and not `m1` (unless a method of `m2` calls a method of `m1`). From the perspective of `m3` and `m4`, `m1` is bound by an inner `ModuleLet`, and so is not bound in the outer scope, that of `m3` or `m4`.

Of course, `m2` cannot call `m3` or `m4`, as the latter two are defined after `m2`.

Module Notation

Just as we did for expressions, we use custom notation to make writing module definitions more natural. First, we give notation for single methods.

```
Notation "'ABSTRACT' 'METHOD' name ( argTy ) : retTy 'USING' uses" :=
  (MethodsNoImpl {| MethArg := argTy; MethRet := retTy; |} name uses)
    (at level 12, name at level 0).

Notation "'METHOD' name ( arg : argTy ) : retTy := c" :=
  (MethodsImpl {| MethArg := argTy; MethRet := retTy; |} name (fun arg ⇒ c))
    (at level 12, name at level 0, arg at level 0).
```

Next, we have notation for modules, which can combine multiple methods together. We use `named_module` simply to wrap a name with a module.

```
Record named_module var c : Type :=
  { NMName : string;
    NMModule : module var c;
  }.

Notation "'MODULE' name {{ submod }}" :=
  ({|  NMName := name; NMModule := submod; |}).

Notation "'LET' m ; kont" :=
  (ModuleLet (NMName m) (NMModule m) kont).

Definition add_module {var c} (nm : named_module var c) :
                    StringMap.t (module var c) -> StringMap.t (module var c) :=
  StringMap.add (NMName nm) (NMModule nm).

Notation "'LETREC' { m1 ; .. ; mN } ; kont" :=
  (ModuleLetRec (add_module m1 .. (add_module mN StringMap.empty) ..) kont).

Notation "'METHODS' { 'STATE' stateTy := initialState ; m1 ; .. ; mN }" :=
  (ModuleState (stateTy := stateTy) initialState (m1 .. (mN MethodsNil) ..)).

Notation "'METHODS' { 'ANYSTATE' ; m1 ; .. ; mN }" :=
  (ModuleAnyState (fun _ ⇒ m1 .. (mN MethodsNil) ..)).
```

3.1.4 Consistency Checks

As alluded to previously, there are some properties of a policy or application that we cannot guarantee by just having Coq type-check the definition. First, we would like to guarantee that every use of a method call type-checks with a previous declaration. Second, we would like to guarantee that there are no name collisions; that is, whenever a module M is defined, there is no other module M' with the same name in scope of M. We have defined several predicates to encapsulate these checks.

To check the first property, `valid_expr menv e` checks that every usage of `MCall` in `e` has the correct type with respect to `menv`. Here, the environment `menv` is a mapping from module names to signatures for the modules in scope of `e`. The only noteworthy case is the `MCall` case, in which the module and method names of the `MCall` are looked up in `menv`, and the resulting signature is compared against the signature of the `MCall`. If either the method called by the `MCall` does not exist or does not have the right type, `valid_expr menv e` will not hold. In the other cases, `valid_expr` just recurses into the subterms of the expression, so we omit them for brevity. We also omit some technicalities related to PHOAS and replace the `var` term in `expr`, `methods`, and `module` with underscores in some of the terms that follow.

```
Definition lookup_method
                    (menv : StringMap.t modulesig) (modname meth : string) :=
  match menv $? modname with
  | Some msig ⇒ msig $? meth
  | None ⇒ None
  end.

Fixpoint valid_expr
         {stateTy ty}
         (menv : StringMap.t modulesig)
         (e : expr _ stateTy ty) : Prop :=
  match e with
  ...
  | @MCall _ _ argTy retTy modname meth arg ⇒
        valid_expr menv arg ∧
              match lookup_method menv modname meth with
              | Some {| MethArg := argTy'; MethRet := retTy'; |} ⇒
```

```
      argTy = argTy' ∧ retTy = retTy'
    | None ⇒ False
    end
  end.
```

`valid_methods` is a natural extension of `valid_expr` to check all the expressions of a `methods` instance.

```
Fixpoint valid_methods
           {stateTy c}
           menv
           (meths : methods _ stateTy c) : Prop :=
  match meths with
  | MethodsNil ⇒ True
  | MethodsNoImpl _ _ _ meths' ⇒ valid_methods menv meths'
  | MethodsImpl _ _ meth meths' ⇒ valid_expr menv (meth _) ∧
                                  valid_methods menv meths'
  end.
```

To extend `valid_methods` to check all of the `MCalls` in a `module`, `valid_module` traverses the module structure and builds up the environment of module names to signatures used in `valid_methods`. `valid_module` also checks that there are no name collisions by requiring that whenever a submodule is declared in a `ModuleLet` or `ModuleLetRec`, its name has not been previously used in the environment.

```
Inductive valid_module : ∀ {c},
           StringMap.t modulesig →
           module _ c → Prop :=
| VMLet {c} menv name submod kont :
    menv $? name = None →
    valid_module menv submod →
    valid_module (menv $+ (name, module_sig submod)) kont →
    valid_module c menv (ModuleLet name submod kont)
| VMLetRec {c} menv submods kont :
    disjoint menv submods →
    (∀ name m,
        submods $? name = Some m →
        valid_module
            (StringMap.update menv (StringMap.map module_sig submods)) m) →
    valid_module
```

```
      (StringMap.update menv (StringMap.map module_sig submods)) kont →
    valid_module c menv (ModuleLetRec submods kont)
| VMAnyState menv meths :
    (∀ stateTy, valid_methods menv (meths stateTy)) →
    valid_module menv (ModuleAnyState meths)
| VMState {c} menv stateTy initial meths :
    valid_methods menv meths →
    valid_module c menv (ModuleState stateTy initial meths).
```

3.1.5 Instantiating Policy Modules

With our policy language defined, we now work towards defining how an application meets a policy. As a first step, we formalize what it means to instantiate a policy module, one that has some unimplemented modules or methods. We instantiate a policy module by "filling in" implementations for the unimplemented methods and modules. Recall that an unimplemented method (given by `MethodsNoImpl`) has a type signature and a whitelist of modules that an implementation of the method can call. The unimplemented method should be filled in by an expression with the same type signature that only makes method calls that respect the whitelist.

`expr_only_uses usable e` checks that `e` only makes method calls respecting the whitelist `usable`. `inst_policy_methods pMeths mMeths` requires that every unimplemented method in `pMeths` has a corresponding implementation in `mMeths` that is valid according to `expr_only_uses`, i.e. that `mMeths` is some filled in version of `pMeths`. An already implemented method can only be filled in by itself, so only the case for an unimplemented method (`InstNoImpl`) is shown below.

```
Fixpoint expr_only_uses
    {stateTy ty}
    (usable : list string)
    (e : expr _ stateTy ty) : bool :=
  match e with
  ...
  | MCall modname _ e' ⇒
      expr_only_uses usable e' &&
      (if List.in_dec modname usable
```

```
      then true
      else false)
  ...
  end.

Inductive inst_policy_methods {stateTy} :
    methods _ stateTy false →
    methods _ stateTy true →
    Prop :=
...
| InstNoImpl : ∀ methtype name pUsable pMeths mMeth mMeths,
    (∀ arg, expr_only_uses pUsable (mMeth arg) = true) →
    inst_policy_methods pMeths mMeths →
    inst_policy_methods
      (MethodsNoImpl methtype name pUsable pMeths)
      (MethodsImpl methtype name mMeth mMeths)
...
```

An unimplemented module may have unimplemented methods or an unspecified state. To fill it in, we must fill in its methods and potentially its state type. `inst_policy_mod p m` requires that the methods of `m` fill in the unimplemented methods of `p`, and in the case where `p` has an unspecified state, requires that `m` has a concrete state type.

```
Inductive inst_policy_mod :
    module _ false →
    module _ true →
    Prop :=
...
| InstAnyState pMeths mStateTy mInitial mMeths :
    inst_policy_methods (pMeths mStateTy) mMeths →
    inst_policy_mod
      (ModuleAnyState pMeths)
      (ModuleState (stateTy := mStateTy) mInitial mMeths)
| InstState stateTy initial pMeths mMeths :
    inst_policy_methods pMeths mMeths →
    inst_policy_mod
      (ModuleState (stateTy := stateTy) initial pMeths)
      (ModuleState initial mMeths).
```

3.2 Semantics

For convenience from now on, we use `CompleteModule` and `IncompleteModule` for `module var true` and `module var false`, respectively.

```
Definition CompleteModule : Type :=
  ∀ var, module var true.

Definition IncompleteModule : Type :=
  ∀ var, module var false.
```

3.2.1 A Big-step Semantics for Applications

To give a semantics for applications, we view them as labeled transition systems. At any point in time, an application's state is comprised of the states of all its modules. A module can be specified by its fully qualified name, represented by `mref`, so we can represent an application's state with `app_state`.

```
Record value : Type :=
  { ValType : type;
    ValValue : interp_type ValType; }.

Definition mref : Type :=
  list string.

Definition app_state : Type :=
  MrefMap.t value.
```

The application transitions when some module runs an expression, potentially changing its state. Running an expression produces a label visible to the outside world, consisting of both the return value of the expression and a list of external method calls made during execution. Recall that external method calls are those made to modules that are not defined in the application.

The judgment `bigstep app stateTy ty m st e tr x st'` means that in the application `app`, upon running the expression `e` from module `m` when the application state is initially `st`, a value of `x` is returned and the application state ends in the state `st'`.

The trace `tr` records the sequence of external method calls made during execution, modeled as a list of instances of `extern`. `bigstep` is a partial semantics; it is not true that every expression can step to a return value according to `bigstep` (e.g. in case of an infinitely looping `While` or a call cycle between modules in a `ModuleLetRec`). Note that `app` is a `CompleteModule`, since it only makes sense to run an application when all its methods are implemented.

`bigstep` has cases for each of the constructors of `expr`, but most of their definitions are straightforward, such as that of `Apply`. For brevity, we describe in more detail only the semantics for `GetState`, `SetState`, and `MCall`, which are the most interesting.

```
Record extern : Type :=
  { ExtModule : string;
    ExtMeth : string;
    ExtArg : value;
    ExtRet : value; }.

Inductive bigstep : ∀ (app : CompleteModule) stateTy ty,
    mref → app_state → expr interp_type stateTy ty → list extern →
    interp_type ty → app_state → Prop :=
...
| StepApply : ∀ m st stateTy ty1 ty2 f e tr x st',
  bigstep app stateTy ty1 m st e tr x st' →
  bigstep app stateTy ty2 m st (Apply f e) tr (f x) st'
| StepGetState : ∀ m st stateTy x,
    MrefMap.MapsTo m x st →
    bigstep app stateTy _ m st GetState nil x st
| StepSetState : ∀ m st stateTy e tr x st',
    bigstep app stateTy _ m st e tr x st' →
    bigstep app stateTy TyUnit m st (SetState e) tr tt (MrefMap.add m x st')
| StepMCallInternal :
    ∀ ...,
    bigstep app stateTy _ caller st arg trArg argX st' →
    name_has_fqmn app caller modname callee →
    lookup_method_on_fqmn app callee methname = Some meth →
    bigstep app stateTy' _ callee st' (meth argX) trMeth methX st'' →
    tr = trArg ++ trMeth →
    bigstep app stateTy _ caller st
                          (MCall argTy retTy modname methname arg) tr methX st''
```

```
| StepMCallExternal :
    ∀ ...,
    bigstep app stateTy _ caller st arg trArg argX st' →
    (∀ fqmn, ¬name_has_fqmn app caller modname fqmn) →
    tr = trArg ++ [{| ExtModule := modname;
                      ExtMeth := methname;
                      ExtArg := {| ValValue := argX; |};
                      ExtRet := {| ValValue := retX; |}; |}] →
    bigstep app stateTy _ caller st
                          (MCall argTy retTy modname methname arg) tr retX st'.
```

When an `MCall` runs, one of two cases can happen: the called module exists in scope of `m`, the calling module; or it does not, and so the call is an external method call. `StepMCallInternal` describes the first case, and `StepMCallExternal` describes the second.

- `StepGetState`: To run `GetState`, we look up the current module's state from `st` and return it.

- `StepSetState`: To run `SetState e`, we first run `e` to a value `x` and new application state `st'`, and then update the module's state to `x` in `st'`. The `SetState` itself should return `tt`.

- `StepMCallInternal`: The call is to a module in scope, so we run the argument to a value, look up the method body to be run and then run the method body. Here, the predicate `name_has_fqmn app caller modname callee` does name resolution; it asserts that in `app` and in the scope of module `caller`, the local name `modname` resolves to the module `callee` (`caller` and `callee` are fully qualified names). From `callee`, `lookup_method_on_fqmn app callee methname` retrieves the body of the method named `methname`.

- `StepMCallExternal`: The call is an external call, so we run the argument to a value and make the external method call by adding an `extern` to the trace. In order for the call to be external, we require that `modname` does not resolve to any module according to `name_has_fqmn`.

We can use `bigstep` to define the trace of a method call.

```
Inductive generates_trace
    (app : CompleteModule) (pmod : mref) (st : app_state) (meth : string) :
    value → list extern → value → app_state → Prop :=
| GeneratesTrace :
  ∀ stateTy methCode arg tr ret st',
    lookup_method_on_fqmn app pmod meth = Some methCode →
    bigstep app stateTy _ pmod st (methCode arg) tr ret st' →
    generates_trace app pmod st meth arg tr ret st'.
```

3.2.2 Trace Equivalence and Policies

Recall that in Chapter 2, we justified that for any policy P, any application indistinguishable (to an outside observer) from an instantiation of P should also satisfy P. An outside observer can only make a sequence of public method calls and see their traces, so we can describe all possible executions of sequences that an observer could see with `possible_execution`. The definition of `trace_equivalence` follows naturally.

```
Definition supertrace := list (mref * string * value * list extern * value).

Inductive possible_execution
              app (allowed : mref → Prop) : supertrace → app_state → Prop :=
| PossibleNil : ∀ st,
  initial_state app st →
  possible_execution app allowed nil st
| PossibleCons : ∀ str pmod st meth arg tr ret st',
  possible_execution app allowed str st →
  allowed pmod →
  generates_trace app pmod st meth arg tr ret st' →
  possible_execution app allowed ((pmod, meth, arg, tr, ret) :: str) st'.

Definition trace_equivalence
              (allowed : mref → Prop) (app1 app2 : CompleteModule) :=
  ∀ supertr,
    (∃ st1, possible_execution (app := app1) allowed supertr st1) ↔
    (∃ st2, possible_execution (app := app2) allowed supertr st2).
```

A module of type `module var false` is actually insufficient to specify a policy, as it does not explicitly say which modules are public or external. Therefore, a fully specified policy should include the module structure (as a `module var false`) as well as all of the public module names and the signatures of the external modules. Finally, we can extend our definition of satisfying a policy: module `m` satisfies the policy `p` when it is trace equivalent with an instantiation `m'` of the policy module. As a sanity check, `m` should also satisfy `valid_module` with respect to the external modules.

```
Record policy : Type :=
  { PolicyModule : IncompleteModule;
    PolicyExtern : StringMap.t modulesig;
    PolicyPublic : list mref }.

Inductive policy_allowed (p : policy) : mref → Prop :=
| PolicyExternAllowed : ∀ name,
    StringMap.In name (PolicyExtern p) →
    policy_allowed p (name :: nil)
| PolicyPublicAllowed : ∀ meth,
    List.In meth (PolicyPublic p) →
    policy_allowed p meth.

Definition satisfies (p : policy) (m : CompleteModule) : Prop :=
  ∃ (m' : CompleteModule),
    trace_equivalence (policy_allowed p) m m' ∧
    valid_module (PolicyExtern p) (m _) ∧
    inst_policy_mod (PolicyModule p _) (m' _).
```

3.3 Comparing Against a Different Design

The current design was not our first design, and a significant amount of time was spent developing a previous version, featuring a different module system design. In this section, we compare the module systems of the two designs, with the purpose of highlighting some advantages and disadvantages of the current design. For the rest of this section, we will call the current design current and the previous design old.

Module Structure in old

```
Definition methodsty := Type → Type.

Inductive module (public internal external : Type) : Type :=
| Public : public → module public internal external
| Internal : internal → module public internal external
| External : external → module public internal external.

Record application_structure
    (public : Type) (publicsig : public → methodsty)
    (external : Type) (externalsig : external → methodsty) : Type :=
  { AppSPublicEqDec : @EqDec public eq eq_equivalence;
    AppSExternalEqDec : @EqDec external eq eq_equivalence;
    AppSInternal : Type;
    AppSInternalEqDec : @EqDec AppSInternal eq eq_equivalence;
    AppSInternalSig : AppSInternal → methodsty;
    AppSModuleCap :
        module public AppSInternal external →
        module public AppSInternal external → Prop;
    AppSPublicState : public → Type;
    AppSPublicInitialState : ∀ (pmod : public), AppSPublicState pmod;
    AppSInternalState : AppSInternal → Type;
    AppSInternalInitialState : ∀ (imod : AppSInternal), AppSInternalState imod;
  }.
```

First, modules are identified with arbitrary types in `Type`. Each module is explicitly designated as public, internal, or external, and each module must have a provided signature. An application's module structure is parametrized by its public and external modules to later provide a type-level guarantee that only two applications with the same public and external modules can be indistinguishable. For technical reasons regarding how method calls work, the types representing modules must also have decidable equality, given by `AppSPublicEqDec`, etc.

In addition to the modules, there is an explicit binary relation `AppSModuleCap` describing the modules' capabilities, where `AppSModuleCap m1 m2` indicates that `m1` has the capability to call `m2`. Finally, public and internal modules have specified states with initial values, given by `AppSPublicState`, etc.

Composability

One very compelling advantage of current over old is the ease with which one can compose and reuse different module definitions.

In old, composing two applications would require combining the two definitions of `public` (as well as `external`, `AppSPublicEqDec`, etc.), likely with a sum type or something similar. If many modules are composed together, `public` (and others) will become a sum type of sum types, and so on, becoming progressively more unwieldy.

Instead, current provides `ModuleLet` and `ModuleLetRec` precisely for composing modules. As a result, two or more modules can be developed separately and be later composed relatively seamlessly. But what if there is a dependency, and one module needs to call a method of the other? If they are developed separately, the calling module does not know the name of the called module or method. We illustrate a solution by example, giving two implementations `modA` and `modB`, where `modB` calls method `methA` of `modA`.

```
Definition modA {c var} : module var c :=
  METHODS {
    STATE TyUnit := tt;
    METHOD "methA" ( _ : TyUnit ) : TyNat := $3
  }.

Definition modB {c var} (modAName methAName : string) : module var c :=
  METHODS {
    STATE TyUnit := tt;
    METHOD "methB" ( _ : TyUnit ) : TyNat :=
      call modAName#methAName ($tt)
  }.

Definition app : module var c :=
  LET MODULE "modA" {{ modA }};
  (modB "modA" "methA").
```

We parametrize `modB` by the name of `modA` and the name of `methA`. This way, `modB` can be written as if it had the capability to call `modA`, even without concrete names. Then, only when combining `modA` and `modB` with a `ModuleLet`, we pass `modB`

the chosen names. This design pattern allows `modA` and `modB` to be implemented independently and combined later, even though `modB` calls a method of `modA`. This pattern is important for modular development, and it is featured prominently in Chapter 5.

Understanding Capabilities

One advantage of old is that it is easier to understand what capabilities each module has. Indeed, the capabilities are explicitly given by `AppSModuleCap`, so anyone who wants to know if module `modA` can call module `modB` can simply consult the `AppSModuleCap` field of an application. Furthermore, `AppSModuleCap` lets developers restrict the capabilities before implementing, protecting them from accidentally violating the intended capabilities in their implementations.

In current, there is no analogue to `AppSModuleCap` that explicitly lists all the capabilities. At best, to know if module `modA` can call module `modB` in application `app`, one can show that there exists a name `name` such that `name_has_fqmn app modA name modB`, as it means that `modB` is in scope of `modA`. However, it may be possible in current, given a binary relation similar to `AppSModuleCap`, to check statically that an implementation complies with the given capabilities.

Infinite Families of Modules

In old, one interesting consequence of having `public`, `external`, and `AppSInternal` live in `Type` is that it is possible to define applications with modules drawn from an infinite family of modules. For example, an application's public modules could be `A`, `B 0`, `B 1`, etc., with one `B` module for every `nat`.

```
Inductive public : Type :=
| A
| B (x : nat).
```

This feature is potentially important for modeling applications that could have an infinite number of modules (e.g. an application with a module for every unique

username).

In current, applications must be of type `module var c`, and so cannot have an infinite number of modules. We can get close, with a function that produces modules of type `module var c`. For example, `infinite_example` implements a family of modules parametrized by a `nat`.

```
Definition infinite_example {var c} (x : nat) : module var c :=
  METHODS {
    STATE TyUnit := tt;
    METHOD "returnValue" ( _ : TyUnit ) : TyNat := $x
  }.
```

This design pattern still allows us to produce members of an infinite family of modules, and it too will appear in Chapter 5.

Chapter 4

Proof Techniques for the User

The definition of `trace_equivalence` from Section 3.2.2 is difficult to prove, as it requires reasoning about the executions of all finite sequences of public method calls. In this chapter, we discuss various definitions and theorems to facilitate proving `trace_equivalence`, based on bisimulation.

4.1 Bisimulation

Bisimulation is a well-known concept in formal methods for proving trace equivalence of two labeled transition systems, as is the case with `trace_equivalence` in our framework. Generally, bisimulation requires giving a state relation and showing that whenever the two systems start in related states and one system transitions, the other can transition in a way that produces the same trace and maintains the relation. Then, if the initial states are related, it is possible to prove trace equivalence.

We have adapted bisimulation in our setting, adding a `public` parameter indicating which modules are public and viewing only public method calls as transitions. Also, since `bigstep` is a partial semantics, we also enforce that for any public method call, if it terminates on one application, then it terminates on the other, as stated by `SimExists1` and `SimExists2`. `bisimulation` allows the prover to prove `trace_equivalence` by reasoning only about the results of single public method calls rather than sequences of calls, and it should simplify the proof effort.

```
Record bisimulation
        (public : mref → Prop)
        (app1 app2 : CompleteModule)
        (Rs : relation app_state) : Prop :=
    { SimPreserved : ∀ st1 st2,
        Rs st1 st2 →
        ∀ pmod,
          public pmod →
          ∀ meth arg tr ret1 st1',
            generates_trace (app := app1) pmod st1 meth arg tr ret1 st1' →
            ∀ ret2 st2',
              generates_trace (app := app2) pmod st2 meth arg tr ret2 st2' →
              ret1 = ret2 ∧ Rs st1' st2';
      SimExists1 : ∀ st1 st2,
          Rs st1 st2 →
          ∀ pmod,
            public pmod →
            ∀ meth arg tr ret1 st1',
              generates_trace (app := app1) pmod st1 meth arg tr ret1 st1' →
              ∃ ret2 st2',
                generates_trace (app := app2) pmod st2 meth arg tr ret2 st2';
      SimExists2 : ∀ st1 st2,
          Rs st1 st2 →
          ∀ pmod,
            public pmod →
            ∀ meth arg tr ret2 st2',
              generates_trace (app := app2) pmod st2 meth arg tr ret2 st2' →
              ∃ ret1 st1',
                generates_trace (app := app1) pmod st1 meth arg tr ret1 st1';
      SimInitial : ∀ st1 st2,
          initial_state (app := app1) st1 →
          initial_state (app := app2) st2 →
          Rs st1 st2; }.

Theorem bisimulation_implies_trace_equivalence : ∀ public app1 app2 Rs,
  bisimulation public app1 app2 Rs →
  trace_equivalence public app1 app2.
```

To prove `trace_equivalence`, we only require that some state relation exists, so we can define that two applications are indistinguishable when there is some bisimulation

relation between them.

```
Definition indistinguishable_on public app1 app2 :=
  ∃ Rs, bisimulation public app1 app2 Rs.

Theorem indistinguishable_implies_trace_equivalence : ∀ public app1 app2,
  indistinguishable_on public app1 app2 →
  trace_equivalence public app1 app2.
```

4.2 Relational Hoare Logic

Bisimulation is a statement at the granularity of method calls, requiring that the states of the two applications satisfy the state relation before and after each public method call. However, it says nothing about states in the middle of execution, and in fact during execution, the states may change to violate the relation temporarily.

To track state changes at the granularity of `bigstep`, we defined a relational Hoare logic for pairs of applications. It allows us to express judgments of the form: if the states of two applications satisfy some precondition P, and the two applications execute (potentially different) expressions according to `bigstep`, then the return values and resulting states satisfy some postcondition Q. The Hoare logic is more general than our definition of bisimulation and more amenable to proof automation, as preconditions and postconditions can be inferred.

We have defined a version of bisimulation based on our relational Hoare logic, `bisim_rhl`, and it has been proven to imply our previous definition of `bisimulation`. We use it to prove bisimulation in some cases in Chapter 5, but its exact definition is not important for the explanations in this thesis.

```
Theorem bisim_rhl_implies_bisim : ∀ public app1 app2 Rs,
  bisim_rhl public app1 app2 Rs →
  bisimulation public app1 app2 Rs.
```

4.3 A Useful Congruence Theorem

One of the primary contributions of this thesis is the following congruence theorem regarding `ModuleLet` and `indistinguishable_on`. Informally, if `m1` and `m2` are indistinguishable (on public modules given by `P`), then the applications `ModuleLet name m1 m3` and `ModuleLet name m2 m3` are also indistinguishable (on public modules given by `Q`), subject to certain conditions on `P` and `Q`. This theorem is a key step in enabling modular verification, as it allows indistinguishability between `m1` and `m2` to be extended to `ModuleLet name m1 m3` and `ModuleLet name m2 m3`, essentially "substituting" `m2` for `m1`.

```
Theorem indistinguishable_on_ModuleLet_sub :
    ∀ (P Q : mref → Prop) name (m1 m2 m3 : CompleteModule),
      P nil →
      (∀ name', Q (name :: name') → P name') →
      indistinguishable_on P m1 m2 →
      indistinguishable_on Q (ModuleLet name m1 m3) (ModuleLet name m2 m3)
```

With this theorem, the intended workflow for proving `trace_equivalence` goals arising from `satisfies` becomes:

- Apply `indistinguishable_implies_trace_equivalence` to change the goal to proving `indistinguishable_on`.
- When possible, apply `indistinguishable_on_ModuleLet_sub` and discharge the extra goals regarding the public modules.
- Prove any remaining `indistinguishable_on` goals by explicitly giving a bisimulation relation and proving either `bisimulation` or `bisim_rhl`.

We will use this workflow to verify some examples in Chapter 5.

Proof Sketch

We directly give a bisimulation relation between the states of `ModuleLet name m1 m3` and `ModuleLet name m2 m3`.

Intuitively, the state of `ModuleLet name m1 m3` can be split into the states of `m1` and `m3`, and the state of `ModuleLet name m2 m3` can be split into the states of `m2` and `m3`. By assumption, the states of `m1` and `m2` are always related by some relation `Rs`. Also, the states of `m3` should be identical, since any call to a module in `m3` should execute identically on both applications. The only place the two executions can diverge is when an `MCall` is made to the module named `name`, since the implementations there differ. But the implementations `m1` and `m2` are indistinguishable, so the executions can also not diverge here. Thus, `ModuleLet name m1 m3` and `ModuleLet name m2 m3` should be in bisimulation, with the relation that the states of `m1` and `m2` are related by `Rs` and that the states of `m3` are always the same.

The predicate `splits_state` formally describes the relationship between `state`, the state of `ModuleLet name submod kont`, and `submod_state` and `kont_state`, the states of `submod` and `kont`, respectively. Then, the relation `ModuleLet_sub_rel name Rs` will be a bisimulation relation between `ModuleLet name m1 m3` and `ModuleLet name m2 m3`.

```
Definition splits_state state name submod_state kont_state : Prop :=
  ∀ mod v,
    MrefMap.MapsTo mod v state ↔
      match mod with
      | nil ⇒ MrefMap.MapsTo mod v kont_state
      | a ::  mod' ⇒
        if eq_dec a name
          then MrefMap.MapsTo mod' v submod_state
          else MrefMap.MapsTo mod v kont_state
      end.

Definition ModuleLet_sub_rel name otherRs state1 state2 :=
  ∃ submod_state1 submod_state2 kont_state,
    splits_state state1 name submod_state1 kont_state ∧
    splits_state state2 name submod_state2 kont_state ∧
    otherRs submod_state1 submod_state2.
```

The remainder of the proof is proving `SimPreserved`, `SimExists1`, `SimExists2`, and `SimInitial`. Proving `SimInitial` is straightforward, albeit somewhat tedious. To prove the other three propositions, we do casework on whether or not `pmod` starts

with `name`. If it does, then the module being called is a public module in both `m1` and `m2`. Roughly, we can connect the method's execution on `ModuleLet name m1 m3` to its execution on `m1`, use indistinguishability to connect it to its execution on `m2`, and then to `ModuleLet name m2 m3`. If `pmod` does not start with `name`, the module being called is in `m3`. Then we can connect the execution on `ModuleLet name m1 m3` to the execution on `m3`, and then directly to `ModuleLet name m2 m3`.

Chapter 5

Case Study: Verifying Simple Database-Backed Applications

To evaluate our framework, we investigate how feasibly it can verify policies for some applications using server-side databases. More specifically, we have devised several guiding examples of applications and policies, spanning several common database security policies. We chose to focus on database security for several reasons:

- Data leaks are a common type of security breach today, and their impact can potentially be mitigated by formally verifying that databases do not leak private data.

- Various methods for database security have been well-studied, providing many examples of interesting policies on which we can use our framework. By showing that our framework supports common policies for databases, we can make a case for the generality of our framework over existing methods.

- Incorrectly implemented database optimizations, such as methods to optimize queries and joins, can introduce unintended security flaws. Our approach seems particularly amenable to proving security of optimizations, since we can view a policy as an unoptimized, more obviously secure version of the application.

5.1 Filtering for Different Users

In our framework, we have verified a simple example application with a database where users can store values and see values stored by other users to compute with. Every value has an additional permission tag, indicating whether it should be public or visible to just the owner. When a value is added to the database, the user provides both the value and the permission that value should have. When a user A requests to see values for some computation, it is returned only the values that user is allowed to access, namely the values owned by A and the public values owned by other users. For simplicity, this application only has two distinct users, `user1` and `user2`, but it is possible to extend this example to more users.

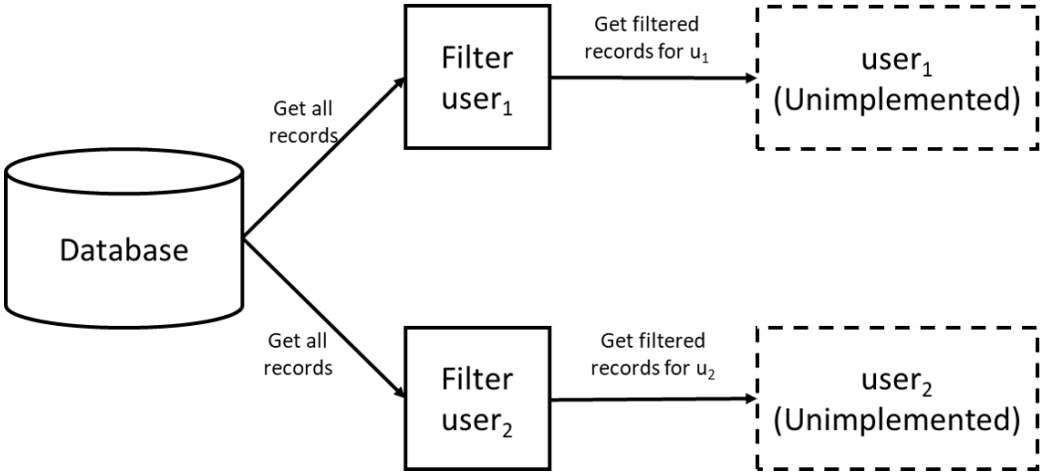

Figure 5-1: A security policy for an application filtering database results based on username

One security policy for this application is shown in Figure 5-1. In the policy, users interact with the application via user modules, each with a unique username. User behavior will not be relevant to security, so each user module's implementation is unspecified in the policy. Each user module only has the capability to request data from the database through its own filter module, which filters based on its associated username. Each filter module has the capability to request values from the central database module, where records of all the values are stored, along with each value's owner and permissions.

Any instantiation of this policy satisfies the following security guarantee: a user module is only ever returned values for which it has permission to access, so that any implementation of a user module cannot compute with data that it should not access. More specifically, the filter module associated with a username s has a specified implementation in the policy that only returns records that the user s can access, and the user module with username s only has the capability to request from that filter module, so that user module can only receive the records it is allowed to access.

An example of a concrete application that we would like to satisfy this policy is an application where a user can request a list of values stored by another user, subject to the same permission restrictions. More precisely, a user u_1 can request a list of values owned by user u_2, and the user module for u_1 should return the list of values that are both owned by u_2 and can be accessed by u_1. It is straightforward to fill in the user implementations in the policy to create such an application, by having each user implementation request the values for which it has permissions, and then further filtering by the desired owner.

5.1.1 Basic Implementation

We showcase some of the implementation details of this application, primarily to highlight design patterns for modular development in our framework.

`basic_database_module` is a simple list-based implementation of the application's database. Note that `basic_database_module` has type `module var c`, so it can be reused in both applications (`module var true`) and policies (`module var false`). Here, `DataRecordSort.sort` uses an implementation of merge sort from the Coq standard library to sort a list of data records by the stored value. We sort the values returned by `getAll` so that different implementations of this module with different internal representations are still indistinguishable.

```
Module DataRecord.
  Definition t : Type := string * nat * bool. (* (owner, val, isPublic) *)
  Definition reified : type := TyTuple (TyTuple TyString TyNat) TyBool.
  ...
End DataRecord.

Definition basic_database_module {c var} : module var c :=
  METHODS {
    STATE (TyList DataRecord.reified) := nil;
    METHOD "getAll" ( _ : TyUnit ) : TyList DataRecord.reified :=
      DataRecordSort.sort ! read;
    METHOD "addRecord" (name_val_isPublic : DataRecord.reified) : TyUnit :=
      write (Cons #name_val_isPublic read)
  }.
```

`filter_module` is an implementation of the filter modules. By parametrizing over `name`, `filter_module` implements the filter modules for any user. `filter_module` should have the capability to access a database module and its methods, and we can pass it those capabilities through `database`, `getAll`, and `addRecord`. Here, the function `DataRecord.can_access name` has type `data_record → bool` and indicates whether the user named `name` can access a given data record.

```
Definition filter_module {c var}
       (name : string) (database getAll addRecord : string) : module var c :=
  METHODS {
    STATE TyUnit := tt;
    METHOD "getAccessibleValues" ( _ : TyUnit ) : TyList DataRecord.reified :=
      plet vals := (call database#getAll ($tt)) in
      (filter (DataRecord.can_access name)) ! #vals;
    METHOD "addRecord" (val_isPublic : TyTuple TyNat TyBool) : TyUnit :=
      plet val      := Fst #val_isPublic in
      plet isPublic := Snd #val_isPublic in
      call database#addRecord (Build (Build $name #val) #isPublic)
  }.
```

`user_filter_policy` connects a filter module to a user module, the latter of which is left unimplemented. Then, we can define `user_filter` (omitted here), an instantiation of `user_filter_policy` that fills in the user module's implementation.

```
Definition user_filter_policy {var}
       (name : string) (database getAll addRecord : string) : module var false :=
  LET MODULE "filter" {{
      (filter_module name database getAll addRecord : module var false)
  }};
  METHODS {
    ANYSTATE;
    ABSTRACT METHOD "getOtherUserValues" ( TyString ) : TyList TyNat
                                                        USING ["filter"];
    ABSTRACT METHOD "addRecord" ( TyTuple TyNat TyBool ) : TyUnit
                                                        USING ["filter"]
  }.
```

Finally, we finish the application definition by connecting a database module to two instances of the user module. To fully specify the policy, `database_app_policy`, we state the external modules (of which there are none) and the public modules (`"user1"` and `"user2"`).

```
Definition database_app_policy_mod {var} : module var false :=
  LET MODULE "database" {{
    (basic_database_module : module var false)
  }};
  LETREC {
    MODULE "user1" {{
      user_filter_policy "user1" "database" "getAll" "addRecord"
    }};
    MODULE "user2" {{
      user_filter_policy "user2" "database" "getAll" "addRecord"
    }}
  };
  METHODS {
    STATE TyUnit := tt;
    METHOD "dummy" (_ : TyUnit) : TyUnit := $tt
  }.

Definition database_app_policy :=
  {| PolicyModule := database_app_policy_mod;
     PolicyExtern := StringMap.empty;
     PolicyPublic := [["user1"]; ["user2"]] |}.
```

We let `database_app` (omitted here) be an instantiation of `database_app_policy_mod` that uses `user_filter` in place of `user_filter_policy`. It is easy to prove that `database_app` satisfies `database_app_policy`, since it is trace equivalent to an instantiation of `database_app_policy_mod`, namely itself.

```
Lemma database_app_trace_equiv_refl :
  trace_equivalence
    (policy_allowed database_app_policy) database_app database_app.

Lemma database_app_valid :
  valid_module StringMap.empty database_app.

Lemma database_app_inst_policy :
  inst_policy_mod database_app_policy_mod database_app.

Theorem database_app_satisfies_policy :
  satisfies database_app_policy database_app.
```

5.1.2 Sharding

Suppose that instead of `basic_database_module`, we wanted to use a different implementation, `sharded_database_module`, that uses a simple sharding scheme. Our scheme simply divides the data into two basic databases based on whether the data's owner's name begins with `"a"`, although the proofs that follow should still hold if we used any other boolean predicate on data records.

```
Definition sharded_database_module {var} : module var true :=
  LETREC {
    MODULE "d1" {{ basic_database_module }};
    MODULE "d2" {{ basic_database_module }}
  };
  METHODS {
    STATE TyUnit := tt;
    METHOD "getAll" ( _ : TyUnit) : TyList DataRecord.reified :=
      plet d1vals := call "d1"#"getAll" ($tt) in
      plet d2vals := call "d2"#"getAll" ($tt) in
      plet allvals :=
        (fun l1_l2 ⇒ (fst l1_l2 ++ snd l1_l2)) ! (Build #d1vals #d2vals))
```

```
    in
    DataRecordSort.sort ! #allvals;
  METHOD "addRecord" (dr : DataRecord.reified) : TyUnit :=
    if_ ((String.prefix "a") ! (DataRecord.getOwner ! #dr))
    then_ { call "d1"#"addRecord" (#dr) }
    else_ { call "d2"#"addRecord" (#dr) }
  }.
```

We get a new application, `sharded_database_app`, by replacing the database implementation in `database_app` with `sharded_database_module`. We would like to prove that `sharded_database_app` also satisfies `database_app_policy`, by showing that it is equivalent to `database_app`.

First, `basic_database_module` and `sharded_database_module` should be indistinguishable, as different implementations of the same module. Indeed, when combined, the two basic databases of `sharded_database_app` should store the same data records as the single database of `database_app`. We can formalize our observation with the relation `sharded_bisim_rel` and use the relational Hoare logic with `sharded_bisim_rel` to prove that the two implementations are indistinguishable.

It is worth noting here that the vast majority of the proof effort in this section is spent proving there is a bisimulation between these two database implementations, as the proof requires much manual reasoning. The rest of the proofs are either trivial or can be easily automated.

```
Definition sharded_bisim_rel (st1 st2 : app_state) :=
  ∀ (l11 l12 l2 : list DataRecord.t),
    MrefMap.find ["d1"] st1 = Some l11 ->
    MrefMap.find ["d2"] st1 = Some l12 ->
    MrefMap.find nil st2 = Some l2 ->
      Permutation (l11 ++ l12) l2.

Lemma database_modules_bisim_rhl :
  bisim_rhl
    (fun mref ⇒ match mref with | nil ⇒ True | _ ⇒ False end)
    sharded_database_module
    (@basic_database_module true)
    sharded_bisim_rel.
```

```
Lemma database_modules_indistinguishable :
  indistinguishable_on
    (fun mref ⇒ match mref with | nil ⇒ True | _ ⇒ False end)
    sharded_database_module
    (@basic_database_module true).
```

Then, using `indistinguishable_on_ModuleLet_sub` from Section 4.3, we can prove that the applications are indistinguishable. From there, it is easy to prove that `sharded_database_app` satisfies `database_app_policy`.

```
Lemma database_apps_indistinguishable :
  indistinguishable_on
    (policy_allowed database_app_policy)
    sharded_database_app
    database_app.

Lemma sharded_database_app_valid :
  valid_module StringMap.empty sharded_database_app.

Lemma database_app_inst_policy :
  inst_policy_mod database_app_policy_mod database_app.

Theorem sharded_database_app_satisfies_policy :
  satisfies database_app_policy sharded_database_app.
```

Chapter 6

Related Work

The problem of formally verifying application security properties is richly studied. In this section, we distinguish our approach from some other prevalent approaches.

6.1 Information-flow Control

One important class of security policies is information-flow control, where the goal is to control how application components access particular pieces of information and ultimately prevent leaking sensitive data.

Static Information-flow Control

A common technique for verifying information-flow control is assigning security classes to both application components and data, then checking by static analysis that each component only gets access to data permitted by the component's security class. In such cases, the security classes typically form a lattice, and a lattice constraint-solving algorithm is used to verify security.

For example, JFlow [5] is an extension to Java that supports static checking of information flow. JFlow allows the types of variables to be annotated with labels, specifying which principals are allowed to read that data. JFlow uses type-inference style derivation rules to derive constraints on the labels and checks that the annota-

tions are consistent with the constraints.

Similarly, Denning and Denning [6] use a lattice-based certification algorithm for their custom programming language. Li and Zdancewic [7] apply typing rules in their web-scripting language so that any well-typed program properly enforces information control. While these techniques work well for applications whose security guarantees can be described by security classes, they are not easily extended to other policies that can be described more naturally without security classes, like channel control. We believe our framework can support more natural descriptions of policies, as demonstrated by Chapter 5.

Dynamic Information-flow Control

The security class approach can also be used for dynamic checking. The LIO monad for Haskell [8] allows data to be dynamically tagged with security class labels from a lattice, which are updated as computation proceeds. During a computation with values in the LIO monad, if there is a violation, an exception is thrown. Dynamic labels in this style have several advantages over static checking, some of which are more fine-grained checking and the ability to compute with labels at runtime.

Crucially however, these dynamic labels come at the cost of performance; extra space and time are required to compute with labels at runtime. Our approach has the potential to reap the benefits of dynamic checking without the performance costs. In our framework, an application without dynamic labels could be shown to satisfy a policy with dynamic labels. Only the application will be run, so it does not incur the cost but still satisfies the security properties inferred from the dynamic labels in the policy.

6.2 Object Capability Systems

Another approach to enforcing security policies is through an object capability system, which is an object-oriented system that restricts access to information by judiciously controlling how object references are used in the system. In capability systems, a

subject A (e.g. an object or module) can only access a resource B (e.g. data storage or another object) when A has a reference to B, which A could have obtained by creating B or getting the reference from a third party C. References are considered unforgeable and are referred to as capabilities. Capability systems have been used to implement secure systems such as KeyKOS [9], a capability-based operating system.

An important property of object capability systems is that subjects do not have ambient authority, and can only interact with resources by demonstrating an explicit capability to do so. A lack of ambient authority mitigates security flaws such as the confused deputy problem, as described by [10]. Our framework also enforces that there is no ambient authority by using the module scoping rules to describe exactly which module interactions are allowed. In both policies and implementations, a module M can only call a method of another module N if M and N are at the exact same module scope. In effect, M is only given capabilities to the modules in the same scope as M. Then, by proving the `valid_module` predicate, a user can verify that every method call made is explicitly authorized by this rule.

Our work does not support policies where capabilties can change dynamically (e.g. a module A grants a capability for C to module B during execution), as each module's capabilities are statically defined in a policy or implementation. We believe this restriction both simplifies verification and makes policies easier to understand for a potential auditor.

6.3 Mediator

Mediator [11] is a language allowing developers to verify semantic equivalence of database-backed applications with different schemas. Similar to our work, Mediator's applications are modeled as transition systems, and Mediator verifies equivalence by finding suitable bisimulation relations. Of note is that Mediator finds and verifies bisimulation relations automatically by calling out to an SMT solver, and that its scope is the general problem of verifying equivalence of databases with different schemas.

Our work encompasses a different scope; we do not formalize or provide automation for this specific problem, although we believe it is possible with our framework to verify concrete instances of this problem (as well as a wide variety of other optimizations). More precisely, within our framework, it is possible to manually write and prove a rewrite rule for a database module stating that two concrete implementations for that module with different schemas are semantically equivalent. We forgo the full automation that Mediator provides in favor of supporting more general optimizations outside of database schema changes.

6.4 UrFlow

UrFlow [12] presents a way for developers to specify policies with a language analogous to standard SQL. To simplify the analysis of these policies, UrFlow is restricted to database-backed web applications written in Ur/Web, also by Chlipala [13]. The policies impose restrictions on what information can be released to a web client and how a database can be updated. The verification process in UrFlow is fully automatic, but the analysis relies on some assumptions about the behavior of these web applications in practice.

One advantage of UrFlow is that developers can verify more flexible, application-specific policies. Policies in UrFlow can reference tables or variables in the application, and so they can more directly describe interactions between different parts of an application. In contrast, security class lattices used in information-flow control cannot directly reference implementation variables.

Similarly, we believe our framework gives more freedom to developers to specify application-specific policies. Policies and implementations are defined with the same language, and we verify policies by proving bisimulation between an implementation and a policy instantiation. As a result, our policies can describe the same behaviors as implementations, going beyond just how information is released to clients. As with Mediator, relative to UrFlow, we sacrifice the ability to completely automatically verify a policy in favor of supporting more general applications and policies.

Chapter 7

Conclusion

In this thesis, we presented our framework in Coq for specifying application security policies and constructing proofs that implementations meet these policies. Our approach frames the task of showing an application meets its policy as one of showing trace equivalence of two applications. The structure of our language enables both modular development and modular verification. With theorems that reduce showing trace equivalence to showing a bisimulation relation, we have verified that a simple database-based application meets a particular policy.

Future Work

There is ample opportunity for improvements and future work on our framework. One area for improvement would be proving additional congruence theorems like the one described in Section 4.3, such as one for substituting in `ModuleLetRec`. More congruence theorems would further enable modular verification in our framework. Another avenue for future work would be to prove the correctness of general optimizations, such as inlining a module and its methods.

Bibliography

[1] The Coq Development Team. The Coq Proof Assistant, version 8.8.0, April 2018.

[2] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A Platform for High-level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.*, 1(ICFP):24:1–24:30, August 2017.

[3] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM.

[4] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ACM Sigplan Notices*, volume 43, pages 143–156. ACM, 2008.

[5] Andrew C. Myers. Jflow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[6] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7):504–513, July 1977.

[7] Peng Li and S. Zdancewic. Practical Information Flow Control in Web-based Information Systems. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 2–15, June 2005.

[8] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. Flexible dynamic information flow control in haskell. In *ACM Sigplan Notices*, volume 46, pages 95–106. ACM, 2011.

[9] Alan C Bomberger, Norman Hardy, A Peri, Frantz Charles, R Landau, William S Frantz, Jonathan S Shapiro, and Ann C Hardy. The keykos nanokernel architecture. In *In Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, 1992.

[10] Norm Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[11] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.*, 2(POPL):56:1–56:29, December 2017.

[12] Adam Chlipala. Static Checking of Dynamically-varying Security Policies in Database-backed Applications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 105–118, Berkeley, CA, USA, 2010. USENIX Association.

[13] Adam Chlipala. Ur/web: A simple model for programming the web. *ACM SIGPLAN Notices*, 50(1):153–165, 2015.