Enforcing Stateful Authorization and Information Flow Policies in FINE

Nikhil Swamy[1] Juan Chen[1] Ravi Chugh[2]

[1] Microsoft Research, Redmond
[2] University of California, San Diego
{nswamy, juanchen}@microsoft.com, rchugh@cs.ucsd.edu

Abstract. Proving software free of security bugs is hard. Languages that ensure that programs correctly enforce their security policies would help, but, to date, no security-typed language has the ability to verify the enforcement of the kinds of policies used in practice—dynamic, stateful policies which address a range of concerns including forms of access control and information flow tracking.
This paper presents FINE, a new source-level security-typed language that, through the use of a simple module system and dependent, refinement, and affine types, checks the enforcement of dynamic security policies applied to real software. FINE is proven sound. A prototype implementation of the compiler and several example programs are available from `http://research.microsoft.com/fine`.

1 Introduction

The security of a well-designed software system often revolves around the concept of a reference monitor, a security-critical kernel that mediates access to resources while enforcing a suitable policy. Reference monitors are expected to be compact and implemented in a form amenable to review. However, increasingly, reference monitors are tasked with enforcing complex policies that simultaneously address various aspects of security, mixing, for example, role- and history-based access control with information flow tracking. Policies are authored separately from the programs they govern, they are composed in non-trivial ways, and, as policies change over time, authorization decisions require reasoning about state. This makes it difficult to establish that a reference monitor enforces a policy correctly.

To illustrate the kinds of security concerns that arise in practice, consider the policy used by CONTINUE [14], a widely used program for managing academic conferences. CONTINUE's security policy is defined using Datalog-like rules in XACML. This policy stands separately from the implementation of the server program, making it hard to connect the policy to the program objects it governs. The policy is also particularly complex in that it makes extensive use of stateful features. For example, the conference management process is staged into a number of phases—in each phase, different policy rules apply. During the submission phase of a conference, authors may submit papers, but this right is revoked after the submission deadline is passed. In the bidding phase, papers are assigned to reviewers after accounting for conflicts of interest. During the rebuttal phase, reviews are disclosed to authors, but care must be taken to ensure that

PC-confidential remarks and scores are not revealed. With such a complex policy to enforce, it is not surprising that the developers of CONTINUE report that almost all the interesting bugs they encountered were related to authorization in some form [7]. Policies used with other kinds of software, such as systems that manage medical records, applications that control the outsourcing of software development, and military systems, arguably have even more complex authorization requirements. Formally verifying that the reference monitors of such systems correctly enforce their policies would help alleviate concerns of security vulnerabilities.

This paper presents FINE, a new source-level security-typed programming language that can be used to implement programs like reference monitors and to check that these programs correctly enforce their security policies. FINE distinguishes itself from prior languages in this line, including FlowCaml [18], Jif [5], Fable [21], Aura [13], and RCF [1], primarily in its ability to express a combination of stateful authorization (none of the prior languages model state) and information flow (which is the focus of FlowCaml and Jif, and can be encoded in Fable and Aura, but not, as far as we are aware, in RCF). The technical contribution of FINE is a new type system (§3) that uses dependent and refinement types to express authorization policies by including first-order logical formulas in the types of program expressions. FINE uses affine types, a weakening of linear types [24], to model changes to the state of an authorization policy. (Variables with an affine type can be used at most once.) The combination of affine and dependent types is subtle and can require tracking uses of affine assumptions in both types and terms. Our formulation keeps the metatheory simple by ensuring that affine variables never appear in types, while still allowing the state of a program to be refined by logical formulas. We also formalize a module system for FINE that provides a simple but strong information-hiding property—we exploit this property to model information flow.

Programming with these advanced typing constructs can impose a significant burden on the programmer. For this reason, languages like Fable and Aura position themselves as intermediate languages because verification depends on intricate security proofs too cumbersome for programmers to write down. Indeed, checking the 2000 lines of code in our benchmark programs produces nearly 200 proof obligations, a proof burden that would overwhelm most programmers. To alleviate this concern, FINE draws on the experience of languages like F7 (an implementation of RCF) and uses Z3 [6], an SMT solver, to automatically discharge proof obligations. The careful combination of refinement and affine types in FINE allows us to use a mature classical prover like Z3. Refinement formulas in FINE only involve the standard logical connectives, avoiding the need for still-experimental linear-logic provers.

We describe our experience using FINE to build several example programs (§4), including a model of the reference monitor of CONTINUE. The complete semantics of FINE, proofs of theorems, and additional examples appear in a technical report [20].

2 FINE, by example

We begin by presenting FINE using several examples. Our first example is a simple form of password-based authentication. Next, we discuss permission-based access control enriched with information flow tracking. Finally, we show how to enforce stateful authorization policies by presenting code examples from our main case-study, a model of the CONTINUE conference management server.

2.1 Authentication, access control, and information flow

FINE's syntax is similar to languages in the ML family. In order to specify and enforce security policies, FINE programmers define modules that provide mediated access to security-sensitive resources. The module Authentication shown below mediates access to authentication routines.

Simple password authentication

```
1 module Authentication
2 type prin = U: string → prin | Admin: prin
3 private type cred :: prin → ⋆ = Auth: p:prin → cred p
4 val login: p:prin → string → option (cred p)
5 let login p pw = if (check_pwd_db p pw) then Some (Auth p) else None
```

The type prin is a standard variant type that represents principal names as either a string for the user's name, or the distinguished constant Admin. The type cred (line 3) is a dependent-type constructor with *kind* prin $\rightarrow \star$, e.g., (cred Admin) is a legal type of kind $\star$ (the kind of normal types, distinguished from the kind of affine types, introduced in §2.2) and represents a credential for the Admin user. Values of the cred p type are constructed using the Auth data constructor. This constructor is given a dependent function type—the argument p is the name of the principal and is in scope to the right of the function arrow. By declaring cred **private**, the Authentication module indicates that its clients cannot directly use the Auth constructor. Instead, the only way a client module can obtain a credential is by calling the login function (given a dependent function type on line 4). The implementation of login (line 5) calls an external function (not shown) to check the password, and, if the password check succeeds, returns a credential for the user p. By indexing cred with the name of the principal which it authenticates, we can statically detect common security errors. For example, a client cannot use login to obtain a credential for U ''Alice'' and later pass it off as a credential for Admin—the type of the former, cred (U ''Alice''), distinguishes it from the latter, which has type cred Admin.

We use Authentication to implement the FileRM module (shown on the next page), a reference monitor that mediates access to a file system. The policies implemented by reference monitors in FINE have two components: the types given to values exposed in the module's interface (e.g., the type of fread on line 7), and policy axioms introduced by the **assume** construct (e.g., **assume** AdminRW on line 6). A security review of a FINE module must confirm that the types and assumptions adequately capture the intent of a high-level policy. Importantly, client code need not be reviewed—typing ensures that clients comply with the reference monitor's security policy.

The FileRM module aims to provide a basic level of access protection on files by ensuring that principals that read and write to files have the requisite permissions. This basic protection is implemented by lines 1-7 of FileRM. The remainder of the module enriches the access control mechanism to track information flows so that, for example, users cannot reveal secrets by copying data from a secret file into a public file.

Permission-based access control and information flow on files

```
 1 module FileRM
 2 open Authentication (* Use non -private symbols from Authentication's namespace *)
 3 (* Propositions and assumptions for file permissions *)
 4 type CanRead:: prin → Sys.file → ⋆
 5 type CanWrite:: prin → Sys.file → ⋆
 6 assume AdminRW: forall f:Sys.file. CanRead Admin f && CanWrite Admin f
 7 val fread_simple: p:prin → cred p → {f:Sys.file | CanRead p f} → string
 8 (* Types and operators to track information flow *)
 9 type label = F : Sys.file → label | J : label → label → label
10 private type tracked :: ∗ → label → ∗ = L : α → p:label → tracked α p
11 val fmap: (α → β ) → l:label → tracked α l → tracked β l
12 val tensor: l:label → m:label → tracked (α → β ) l → tracked α m → tracked β (J l m)
13 (* Types and axioms for a partial order on labels *)
14 type CanFlow:: label → label → ⋆
15 assume Lattice: forall l:label, m1:label, m2:label. (CanFlow l l) &&
16    ((CanFlow l m1 && CanFlow l m2) ⇒ CanFlow l (J m1 m2)) &&
17    ((CanFlow m1 l && CanFlow m2 l) ⇒ CanFlow (J m1 m2) l)
18 assume Atomicflow: forall f:Sys.file, g:Sys.file.
19   (forall p:prin. CanRead p g ⇒ CanRead p f) ⇒ CanFlow (F f) (F g)
20 (* Secure wrappers for system calls *)
21 val fread: p:prin→ cred p→ f:{x:Sys.file | CanRead p x}→ tracked string (F f)
22 let fread p c f = L (Sys.fread f) (F f)
23 val fwrite: p:prin→ cred p→ f:{x:Sys.file | CanWrite p x}→
24                 l:{y:label | CanFlow y (F f)} → tracked string l → unit
25 let fwrite p c f l (L s x) = Sys.fwrite f s
```

FileRM defines dependent-type constructors CanRead and CanWrite to describe access permissions. Permissions are granted using assumptions like AdminRW, which states that the Admin user has read- and write-permissions on all files. Client programs can use axioms like AdminRW to produce evidence of the propositions required to call functions like fread_simple, which wrap the underlying system calls. Client programs are assumed to not have direct access to these system calls—this can be established using standard systems techniques like sandboxing [25]. The type of fread_simple is used to enforce an access control policy. A caller of fread_simple is required to pass in a credential for a user p and a file handle f, where f has the refined type {x:Sys.file | CanRead p x} indicating that p has permission to read f.

We used fread_simple mainly to illustrate how refinement types can express simple authorization policies. When leaks due to information flows are a concern, FileRM would not include fread_simple in the API exposed to client programs. Clients would have to use fread instead, which augments fread_simple with information flow controls.

The encoding of information flow shown in FileRM is based on a model developed with the Fable calculus [21]. Information flow policies are specified and enforced by tagging sensitive data with *security labels* that record provenance. The type label (line 9) represents the provenance of data derived from one or more files, F x for data from file x, and J l1 l2 for data derived from the files in both l1 and l2. The dependent-type constructor tracked associates labels with data. For example, tracked string (F x) represents a string that originated from the file x. Importantly, tracked is defined as a private type. Client programs can only manipulate tracked values using functions that appear in the interface of FileRM, e.g., fmap, a functor that allows functions to be lifted into the tracked type and tensor, a combinator that treats the tracked type as an indexed applicative functor. Prior work on Fable showed that encodings of this style can be proved to correctly enforce security properties like noninterference.

Next, we define a type CanFlow and assumptions to describe a partial order on labels. The Lattice assumption states that the J constructor behaves as the least-upper-bound relation on a join semi-lattice and that flows are permissible from lower labels to higher ones. The Atomicflow assumption states that data can flow from a file f to a file g only if all principals that can read g can also read f. The types of fread and fwrite use these constructs to track information flow. The type of fread shows that the content of f is returned as a string tagged with its provenance, i.e., tracked string (F f). The type of fwrite requires that the string written to a file f has provenance l, where the refinement CanFlow y (F f) on the type of l requires it to only contain data visible to the readers of f.

Specific file permissions and a client program

```
 1 open Authentication, FileRM
 2 assume R_a: CanRead (U ‘‘Alice’’) ‘‘a.txt’’ &&
 3       (forall p:prin.CanRead p ‘‘a.txt’’ ⇒ p=U ‘‘Alice’’ || p=Admin)
 4 assume R_ab: CanRead (U ‘‘Alice’’) ‘‘ab.txt’’ && CanRead (U ‘‘Bob’’) ‘‘ab.txt’’ &&
 5       (forall p:prin.CanRead p ‘‘ab.txt’’ ⇒ p=U ‘‘Alice’’ || p=U ‘‘Bob’’ || p=Admin)
 6 val strcat: string → string → string
 7 let sudo (c:cred Admin) =
 8   let a, ab = fread Admin c ‘‘a.txt’’, fread Admin c ‘‘ab.txt’’ in
 9   let a_ab = tensor (F ‘‘a.txt’’) (F ‘‘ab.txt’’) (fmap strcat (F ‘‘a.txt’’) a) ab in
10       fwrite Admin c ‘‘a.txt’’ (J (F ‘‘a.txt’’) (F ‘‘ab.txt’’)) a_ab
```

Additional policy assumptions and client code. The code sample above includes axioms R_a and R_ab to define access permissions for some files. (We assume here that Sys.file and string are synonyms.) We also show a client program,sudo, which runs with the credentials of Admin, concatenates data from files a.txt and ab.txt, and writes the result to the file a.txt. In addition to Admin, the file a.txt is readable only by the user Alice and ab.txt only by Alice and Bob. Thus, sudo is secure since it writes to a.txt data that can be read by Alice and Admin. In contrast, if sudo were to write the result to ab.txt, the contents of a.txt are leaked to Bob, and this program should be detected as insecure.

At each call to fread, the solver appeals to AdminRW to show that Admin has read permission on the files. To concatenate tracked strings, we use the fmap and tensor operators from the FileRM API.[3] The type of a_ab is tracked string (J (F ‘‘a.txt’’) (F ‘‘ab.txt’’)). At line 10, we need to prove CanFlow (J (F ‘‘a.txt’’) (F ‘‘ab.txt’’)) (F ‘‘a.txt’’), which is discharged automatically by Z3. Trying to write a_ab to ab.txt instead results in a type error.

[3] Our implementation currently lacks support for implicit parameters in function calls. Defining all label parameters to be implicit would produce more terse programs. For example, concatenation of tracked strings would read tensor (fmap strcat a) ab.

2.2 Stateful authorization in the CONTINUE conference manager

We now present a more substantial example in FINE: a model of the CONTINUE conference management server. We first present a reference monitor ConfRM which mediates access to a database of paper submissions and reviews. Next, we show ConfPolicy, a set of policy axioms used to configure the reference monitor. Finally, we discuss ConfWeb, a web server processing requests and accessing the database via the reference monitor.

A model of stateful authorization. The design of the ConfRM reference monitor is based on a framework due to Dougherty et al. [7] for reasoning about the correctness of Datalog-style dynamic policies. This model specifies policies as inference rules that derive permissions from basic authorization attributes. For example, attributes may include assertions about a principal's role membership or the phase of the conference, and inference rules could grant permissions to principals depending on the current phase and role activations. Over time, whether due to a program's actions or due to external events, the set of authorization attributes can change. For example, to access a resource, a principal may alter the state of the authorization policy by activating a role; or, the PC chair can change the phase of the conference. In this state, the policy may grant a specific privilege to the principal, but a subsequent role deactivation revokes the privilege. Dougherty et al. show that this model captures many common policies and can be used to reason about policy correctness.

This model of stateful authorization can be represented directly in FINE. The type st represents the set of basic authorization attributes (line 10 in the listing on the next page). Attributes include values like Role (U ''Alice'') Author to represent a role activation, or values like Assigned r p to indicate that a paper p has been assigned to a reviewer r. The type perm represents permissions (the relations derived using inference rules from the basic authorization attributes). For example, Permit (U ''Alice'') (Submit p) represents a permission granted to an author. ConfRM also defines two propositions for stating invariants about the current state of the policy. Line 12 shows the type In, a proposition about list membership, e.g., In a s states that a is a member of the list s. We elide standard assumptions that axiomatize list membership, but show a simple recursive function check that decides list membership (line 13-15). The proposition Derivable s p (line 16) asserts that a permission p is derivable from the collection of authorization attributes s. We define two type abbreviations for refinements of the st type: rst<p> are those states in which p is derivable, and inst<a> are those states that include a.

For a flavor of refinement type checking, consider the check function. The essence of typing this function is proving that the true sub-expression can be given the type {b:bool | In a l}. We accomplish this by typing the value true in a context that records equalities between l and hd::tl (induced by the pattern match); an assumption that the expression (equals a hd) has the type {b:bool | b=true ⇔ a=hd} (by a type given to the built-in equals operator); an assumption that (equals a hd) evaluates to true (since we are typing the **then**-branch); and the axioms for list membership. We determine if the goal (In a l) is deducible from the assumptions by including the negation of the goal among the assumptions and requiring the solver to prove the resulting theory unsatisfiable.

Modeling state updates with affine types. The type constructor StateIs (line 19) addresses two concerns. A value of type StateIs s represents an assertion that s contains the current state of authorization facts. ConfRM uses this assertion to ensure the integrity of its authorization facts. StateIs is declared private, so untrusted clients cannot use the Sign constructor to forge StateIs assertions. Moreover, since the authorization state can change over time, FINE's type system provides a way to revoke StateIs assertions about stale states. For example, after a reviewer r has submitted a review for a paper p, we may add the fact Reviewed r p to the set of authorization facts s, revoke the assertion StateIs s, and use StateIs ((Reviewed r p)::s) instead.

A fragment of a reference monitor for a conference management server

```
 1 module ConfRM
 2 open Authentication
 3 type role = Author | Reviewer | Chair
 4 type phase = Submission | Reviewing | Meeting
 5 type paper = {id:int; title:string; author:prin; contents:string}
 6 type attr = Role : prin → role → attr | Assigned : prin → paper → attr
 7            | Phase : phase → attr | Reviewed : prin → paper → attr
 8 type action = Submit: paper → action | Review: paper → action
 9              | ReadScore: paper → action | CloseSub: action
10 type st = list attr
11 type perm = Permit : prin → action → perm
12 type In :: attr → st → ⋆
13 val check: a:attr → l:st → {b:bool | b=true ⇒ In a l}
14 let rec check a l = match l with [] → false
15                                | hd::tl → if equals a hd then true else check a tl
16 type Derivable :: st → perm → ⋆
17 type rst<p:perm> = {s:st | Derivable s p}
18 type inst<a:attr> = {s:st | In a s}
19 private type StateIs:: st → A = Sign: s:st → StateIs s
20 val submit: q:prin→ cred q→ p:paper→ s:rst<Permit q (Submit p)>→ StateIs s→ StateIs s
21 val review: r:prin → cred r → p:paper → q:string → s:rst<Permit r (Review p)> →
22              StateIs s → (s':inst<Reviewed r p> ∗ StateIs s')
23 val close_sub: c:prin → cred c → s:rst<Permit c CloseSub> →
24              StateIs s → (s':inst<Phase Reviewing> ∗ StateIs s')
```

FINE types are classified into two basic kinds: $\star$, the kind of normal types, and A, the kind of affine types. By declaring StateIs :: st → A we indicate that StateIs constructs an affine type from a value of type st. When the state of the authorization policy changes from s to t, ConfRM constructs a value Sign t to assert StateIs t, while destructing a StateIs s value to ensure that the assertion about the stale state s can never be used again.

An external API to the conference DB. Lines 20-24 show the types of functions exposed by ConfRM to clients. Using the refined state type rst<p>, the API ensures that each function is only called in states where the permission p is derivable. The submit function requires Permit q (Submit p) to be derivable in the state s. By returning StateIs s, the type of submit indicates that it does not change the authorization state. The review function allows a reviewer r to submit a review and then changes the authorization state to record the submission. The return type of review is a dependent pair consisting of a new list of authorization attributes s', and an assertion of type StateIs s' to indicate that s' is the new authorization state. The close_sub function has a similar type and allows the program chair to change the phase of the conference.

An example policy and a main event loop for the server

```
 1 module ConfPolicy : ConfRM
 2 let init:(s:st ∗ StateIs s) = let a = [Role (U ‘‘Andy’’) Chair; ...] in (a, Sign a)
 3 assume C1: forall (q:prin), (p:paper), (s:st).
 4   In (Phase Submission) s && In (Role q Author) s ⇒ Derivable s (Permit q (Submit p))
 5 assume C2: forall (r:prin), (p:paper), (s:st).
 6   In (Phase Reviewing) s && In (Assigned r p) s ⇒ Derivable s (Permit r (Review p))
 7 assume ...
 8 (∗ Main event loop ∗)
 9 module ConfWeb
10 open Authentication, ConfRM, ConfPolicy
11 let rec loop s = match get_request() with
12   | Submit_paper q credq paper → let (a,tok) = s in
13       if (check (Phase Submission) a) and (check (Role q Author) a) then
14         let s1 = submit q credq paper a tok in
15         let _ = resp ‘‘Thanks for your submission!’’ in loop (a, s1)
16       else let _ = resp ‘‘Submissions are closed, or you are not an author.’’ in loop (a,tok)
17   | Submit_review r credr paper review → ...
18 let _ = loop ConfPolicy.init
```

A sample policy. The module ConfPolicy above configures the ConfRM reference monitor with policy assumptions. At line 2, we show init, an initial collection of authorization attributes a, signed to attest that a is the authorization state. The Sign data constructor requires the privilege of ConfRM—FINE's module system grants this privilege to ConfPolicy using the notation **module** ConfPolicy : ConfRM, which allows ConfPolicy to use the private constructors of ConfRM. The assumptions C1-C2 show how permissions can be derived from authorization attributes—different conferences can use the same ConfRM but get different enforcement semantics by using different policy files.

An event loop to handle web requests. Finally, we show fragments from ConfWeb, a program that handles web requests to the conference management site. The main event loop of ConfWeb waits for a request (type elided). If principal q wishes to submit a paper, we check that the conference is in the Submission phase, and that q is registered in the role of an Author. We give the built-in boolean operator and the type x:bool → y:bool → {z:bool | z=true ⇔ x=true && y=true}. We can use this type, the type of check, and assumption C1, to refine the type of the current state a in the **then**-branch to rst<Permit q (Submit paper)>.

2.3 Elements of FINE that enable stateful programming

Before proceeding to a formal semantics for FINE, we discuss a number of elements in the design of FINE that facilitate, and in some cases simplify, stateful programming.

Non-affine state simplifies programming. Programming with affine types can be difficult, since affine variables can never be used more than once. Our approach of using an affine assertion StateIs s to track the current authorization state minimizes the difficulty. Importantly, the collection of authorization facts s is itself not affine and can be freely used several times, e.g., s is used in several calls to check. Non-affine state also enables writing functions like check, which, if s was affine, would destroy the state of the program. Only the affine token, tok:StateIs s, must be used with care, to ensure that it is not duplicated.

Non-affine refinements simplify automated proofs. Even ignoring the inability of prior languages to handle stateful policies, the proof terms required for our examples in languages like Fable or Aura would be extremely unwieldy. By ensuring that refinement formulas always apply to non-affine values, our proof system is kept tractable, allowing us to use Z3 to automatically discharge proof obligations. A naïve combination of dependent and affine types would allow refinements to apply to affine values, necessitating an embedding of linear logic in Z3. Our approach avoids this complication, while retaining the ability to refine the changing state of a program with logical formulas.

Affine types enable flexible mixing of stateful and pure code. Another approach to working with stateful policies could be to use an abstract monad. FINE's module system certainly supports programming in this style. However, affine types afford greater flexibility. For example, rather than monadically threading a monolithic store through the program, FINE programs can partition the state and pass only the relevant parts of the store to functions that need it. We use this idiom to good effect in one of our benchmark programs (FileAutomaton in §4), in which a bit of state representing the current state of a file is associated with the file handle rather than using a monolithic store to maintain the state of all file handles. Another benchmark, a model of an email client, uses affine types to model capabilities [15] that grant programs restricted access to certain sensitive stateful operations, such as sending emails.

3 Formalizing FINE

Our compiler translates FINE programs in type-preserving manner to .NET bytecode (CIL) [8]. Although we do not report on our type-preservation results in this paper, this design plays a significant role in various aspects of FINE's type system. This section formalizes FINE, presents a soundness result for the type system, and an information-hiding property for the module system. We begin by presenting a core syntax for FINE.

3.1 Core syntax

Our formulation of FINE's module system is based on Grossman et al's [11] syntactic approach to type abstraction. In this formulation, module names correspond to "principals" and are ranged over by the meta-variables p, q, and r. Source expressions are annotated with the names of the modules to which they belong—in the form $\langle e \rangle_p$, the expression e delimited within brackets is privileged to use p's private types concretely. A principal constant is denoted p, and we include two distinguished principals: $\top$ includes the privileges of all other principals, and $\bot$ has no privileges. Values are partitioned into families corresponding to principals. A pre-value for code with p-privilege, u_p, is a variable or a fully-applied data constructor D. Values for p are either its pre-values, abstractions, or pre-values u_q for some other principal q, enclosed within brackets to denote that u_q carries q-privilege. The dynamic semantics of FINE (§3.3) tracks the privilege associated with an expression using these brackets and allows us to prove (§3.4) that programs without p-privilege treat p-values abstractly.

Core syntax of FINE

$p, q, r ::= \mathsf{p} \mid \top \mid \bot$	principals
$u_p ::= x \mid D\ \bar{\tau}\ \bar{v_p}$	pre p-values
$v_p ::= u_p \mid \lambda x{:}\tau.e \mid \Lambda\alpha{::}\kappa.e \mid \langle u_q \rangle_q$	p-values
$e ::= v_p \mid$ let $x = e_1$ in $e_2 \mid$ fix $f{:}\tau.e \mid v_p\ v_q \mid v_p\ \tau \mid \langle e \rangle_p$	terms
match v_p with $D\ \bar{\tau}\ \bar{x} \to e_1$ else e_2	
$\tau, \phi ::= \alpha \mid x{:}\tau \to \tau' \mid \forall\alpha{::}\kappa.\tau \mid \{x{:}\tau \mid \phi\} \mid\ !\tau \mid T \mid \tau\ \tau' \mid \tau\ v_p$	types
$\kappa ::= \star \mid \mathtt{A} \mid \star \to \kappa \mid \mathtt{A} \to \kappa \mid \tau \to \kappa$	kinds
$S ::= T{::}\kappa \mid D{:}(p, \tau) \mid p \sqsubseteq q \mid S, S' \mid \cdot$	signature
$\Gamma ::= \alpha{::}\kappa \mid x{:}(p, \tau) \mid v_p \doteq v'_p \mid \Gamma, \Gamma' \mid \cdot$	type env.

Expressions e are standard for a polymorphic lambda calculus. Types τ include dependent function types $x{:}\tau \to \tau'$, where x names the formal parameter and is bound in τ'. Polymorphic types $\forall\alpha{::}\kappa.\tau$ decorate the abstracted type variable α with its kind κ. Refinement types are written $\{x{:}\tau \mid \phi\}$, where ϕ is a type in which x is bound. An affine qualifier can be attached to a type using $!\tau$. Type constructors T can be applied to other types using $\tau\ \tau'$ or terms using $\tau\ v_p$. Note that type-level terms are always values, not expressions—this restriction explains our use of A-normal form [10] for the expression language. This form allows every intermediate result to be named and for these names to appear, potentially, as type indices. Types are partitioned into normal types (kind $\star$) and affine types (kind $\mathtt{A}$). Type constructors T construct types of kind κ from normal types ($\star \to \kappa$), affine types ($\mathtt{A} \to \kappa$), or τ-typed terms ($\tau \to \kappa$). Although included in our implementation, for simplicity, our formalization omits dependent pairs.

Desugaring FINE **modules.** The type and data constructor declarations in a FINE module are desugared to a signature S. The type constructors of the Authentication module of §2.1, for example, are desugared to prin::$\star$ and cred::prin $\to \star$. Data constructors D are associated their type, as well as the privilege p required for their use. For example, the constructors of the prin type are U:$(\bot,$ string $\to$ prin$)$ and Admin:$(\bot,$ prin$)$, indicating that these may be used freely in unprivileged code. In contrast, being declared **private**, the constructor of the cred type is desugared to Auth : (Authentication, p:prin $\to$ cred p), indicating that it may only be used in code marked with the privilege of the Authentication module. Additionally, signatures use $p \sqsubseteq q$ to record a partial order among principals, with $\bot \sqsubseteq p \sqsubseteq \top$, for all p. We use this to represent sharing between modules, as achieved by the ConfPolicy : ConfRM declaration from §2.2. This is translated to the relation ConfRM $\sqsubseteq$ ConfPolicy, to indicate that ConfPolicy holds the privileges of ConfRM (and, in particular, can use ConfRM's private data constructors).

Desugaring formulas and assumptions. Refinement formulas and assumptions are represented using type and data constructors, respectively. For example, we use type constructors like And::$\star \to \star \to \star$ to represent the logical connectives. We model equality by specializing it to each type, e.g., Eq_bool::bool $\to$ bool $\to \star$. A polymorphic treatment of equality poses no fundamental difficulty, but we use a monomorphic treatment here for simplicity. Quantification is represented using the binders in dependent functions and pairs. For example, the AdminRW assumption from §2.1 is desugared to AdminRW : $(\bot,$ f:file $\to$ And (CanRead Admin f) (CanWrite Admin f)$)$. Note that assumptions are always public—we leave an exploration of private assumptions to future work.

Well-formedness conditions on data constructors. The soundness of FINE's type system relies on some restrictions on the use of data constructors D. We mention these restrictions briefly here, but space constraints leave their formalization and further discussion to a technical report [20]. First, we disallow partial application of data constructors as this complicates our translation to CIL. Next, we require the type of each data constructor to be of the form: $\forall\bar{\alpha}{::}\bar{\kappa}.x_1{:}\tau_1 \to \ldots \to x_n{:}\tau_n \to \tau$, i.e., we require any type arguments to precede any term arguments, although each term argument $x_i{:}\tau_i$ may itself contain quantifiers. This restriction is merely a convenience—it simplifies the shape of our pattern matching constructs. Finally, for each data constructor D with a type as shown above, we require $\bar{\alpha} \subseteq$ Free-type-variables(τ), i.e., every type argument must appear as an index on the constructed type τ. This is a more significant restriction and is necessary for showing that well-typed programs enjoy a type-erasure property.

3.2 Static semantics

The static semantics makes use of a typing environment Γ, which binds type and term variables, and records the results of pattern matching tests using $v_p \doteq v'_p$. Variables x, like data constructors, are associated with a principal p representing the privilege required for their use.

Well-formedness of kinds: $S \vdash_i k$**, and kinding of types:** $S; \Gamma \vdash \tau :: \kappa$
Where, $i ::= \cdot \mid 1$, and $\star \leq \star,\ \mathtt{A} \leq \mathtt{A},\ \star \leq \mathtt{A}$

$$\frac{}{S \vdash_. \star} \qquad \frac{}{S \vdash_i \mathtt{A}} \qquad \frac{S \vdash_i \kappa}{S \vdash_i \star \to \kappa} \qquad \frac{S \vdash_1 \kappa}{S \vdash_i \mathtt{A} \to \kappa} \qquad \frac{S; \cdot \vdash \tau :: \star \quad S \vdash_i \kappa}{S \vdash_i \tau \to \kappa}$$

$$\frac{}{S; \Gamma \vdash \alpha :: \Gamma(\alpha)}\text{(K1)} \qquad \frac{}{S; \Gamma \vdash T :: S(T)}\text{(K2)} \qquad \frac{S; \Gamma \vdash \tau :: \star}{S; \Gamma \vdash !\tau :: \mathtt{A}}\text{(K3)}$$

$$\frac{\begin{array}{c}S; \Gamma, \alpha{:}\kappa \vdash \tau :: \kappa' \\ \kappa, \kappa' \in \{\star, \mathtt{A}\}\end{array}}{S; \Gamma \vdash \forall\alpha{::}\kappa.\tau :: \star}\text{(K4)} \qquad \frac{\begin{array}{c}S; \Gamma \vdash \tau_1 :: \kappa \quad \kappa \leq \kappa' \\ S; \Gamma, x{:}(p, \tau_1) \vdash \tau_2 :: \kappa'\end{array}}{S; \Gamma \vdash x{:}\tau_1 \to \tau_2 :: \star}\text{(K5)} \qquad \frac{\begin{array}{c}S; \Gamma \vdash \tau_1 :: \kappa' \to \kappa \\ S; \Gamma \vdash \tau_2 :: \kappa'\end{array}}{S; \Gamma \vdash \tau_1\ \tau_2 :: \kappa}\text{(K6)}$$

$$\frac{S; \Gamma \vdash \tau_1 :: \tau \to \kappa \quad S; \Gamma; \cdot \vdash_\top v_p : \tau}{S; \Gamma \vdash \tau_1\ v_p :: \kappa}\text{(K7)} \qquad \frac{S; \Gamma \vdash \tau :: \star \quad S; \Gamma, x{:}(p, \tau) \vdash \phi :: \star}{S; \Gamma \vdash \{x{:}\tau \mid \phi\} :: \star}\text{(K8)}$$

The first judgment $S \vdash_i \kappa$, shown above, defines a well-formedness relation on kinds. This judgment establishes two properties. First, types constructed from affine types must themselves be affine—this is standard [24]. Without this restriction, an affine value can be stored in a non-affine value and be used more than once. To enforce this property, we index the judgment using $i ::= \cdot \mid 1$, and when checking a kind $\mathtt{A} \to \kappa$, we require κ to finally produce an $\mathtt{A}$-kinded type. The second restriction, enforced by the first premise ($S; \cdot \vdash \tau :: \star$) of the last rule, ensures that only non-affine values appear in a dependent type. Note that we omit higher kinds (e.g., $(\star \to \star) \to \star$) as these are not easily translated to CIL.

The judgment $S; \Gamma \vdash \tau :: \kappa$ states that τ has kind κ. Types inhabited by terms always have kind $\star$ or $\mathtt{A}$. (K3) rules out "doubly-affine" types ($!!\tau$). (K4) allows abstraction only over $\star$ and $\mathtt{A}$-kinded types. (K5) requires that the type τ_1 of a function's parameter always have kind $\star$ or $\mathtt{A}$ and that functions with affine arguments produce affine results, both captured by an auxiliary relation on kinds, $\kappa \leq \kappa'$. (K7) checks the well-formedness of dependent types. As in Aura and RCF, we restrict type-level terms to values e.g., Eq_bool (true && false) false is not a well-formed type. This restriction reduces expressiveness by ruling out type-level computations, but greatly simplifies the compilation to CIL. The second premise of (K7) uses the typing judgment—we describe it shortly. (K8) only allows non-affine types τ to be refined by non-affine formulas ϕ.

Expression typing: $S; \Gamma; X \vdash_p e : \tau$
Where, $X ::= \cdot \mid x, X$; $\quad Q(X, \tau) = !\tau$, $Q(\cdot, \tau) = \tau$; $\quad$ and $?\tau$ denotes τ or $!\tau$

$$\frac{S(D) = (p, \tau)}{S; \Gamma; \cdot \vdash_p D : \tau}\;\text{(T1)} \qquad \frac{\Gamma(x) = (p, \tau) \qquad S; \Gamma \vdash \tau :: \star}{S; \Gamma; \cdot \vdash_p x : \tau}\;\text{(T2)} \qquad \frac{\Gamma(x) = (p, \tau)}{S; \Gamma; x \vdash_p x : \tau}\;\text{(T3)}$$

$$\frac{q \sqsubseteq p \in S \qquad S; \Gamma; X \vdash_q e : \tau}{S; \Gamma; X, X' \vdash_p e : \tau}\;\text{(T4)} \qquad \frac{S; \Gamma \vdash \tau :: \star \qquad S; \Gamma, f{:}(p, \tau); \cdot \vdash_p v_p : \tau}{S; \Gamma; \cdot \vdash_p \text{fix } f{:}\tau.v_p : \tau}\;\text{(T5)}$$

$$\frac{\begin{array}{c} S; \Gamma \vdash \tau_1 :: \kappa \qquad \kappa \in \{\star, \mathtt{A}\} \\ S; \Gamma, x{:}(p, \tau_1); X, x \vdash_p e : \tau_2 \end{array}}{S; \Gamma; X \vdash_p \lambda x{:}\tau_1.e : Q(X, x{:}\tau_1 \to \tau_2)}\;\text{(T6)} \qquad \frac{\begin{array}{c} \kappa \in \{\star, \mathtt{A}\} \\ S; \Gamma, \alpha{::}\kappa; X \vdash_p e : \tau' \end{array}}{S; \Gamma; X \vdash_p \Lambda\alpha{::}\kappa.e : Q(X, \forall\alpha{::}\kappa.\tau')}\;\text{(T7)}$$

$$\frac{\begin{array}{c} S; \Gamma; X \vdash_p e_1 : \tau_1 \qquad S; \Gamma \vdash \tau_2 :: \kappa \\ S; \Gamma, x{:}(p, \tau_1); X', x \vdash_p e_2 : \tau_2 \end{array}}{S; \Gamma; X, X' \vdash_p \text{let } x = e_1 \text{ in } e_2 : \tau_2}\;\text{(T8)} \qquad \frac{\begin{array}{c} S; \Gamma; X \vdash_p v_q : ?x{:}\tau_1 \to \tau_2 \\ S; \Gamma; X' \vdash_p v_r : \tau_1 \end{array}}{S; \Gamma; X, X' \vdash_p v_q\ v_r : \tau_2[v_r/x]}\;\text{(T9)}$$

$$\frac{S; \Gamma; X \vdash_p v_q : ?\forall\alpha{::}\kappa.\tau \qquad S; \Gamma \vdash \tau' :: \kappa}{S; \Gamma; X \vdash_p v_q\ \tau' : \tau[\tau'/\alpha]}\;\text{(T10)} \qquad \frac{S; \Gamma; X \vdash_q e : \tau}{S; \Gamma; X \vdash_p \langle e \rangle_q : \tau}\;\text{(T11)}$$

$$\frac{\begin{array}{c} S; \Gamma; X \vdash_p v_q : \tau' \qquad S; \Gamma, \bar{x}{:}(\bar{p}, \bar{\tau_x}); \bar{x} \vdash_p D\ \bar{\tau}\ \bar{x} : \tau'' \qquad S; \Gamma \vdash \text{unify}(\tau', \tau'') : \bar{x} \doteq \bar{v} \\ S; \Gamma, \bar{x}{:}(\bar{p}, \bar{\tau_x}), \bar{x} \doteq \bar{v}, v_q \doteq D\ \bar{\tau}\ \bar{x}; X', \bar{x} \vdash_p e_1 : \tau \qquad S; \Gamma; X' \vdash_p e_2 : \tau \end{array}}{S; \Gamma; X, X' \vdash_p \text{match } v_q \text{ with } D\ \bar{\tau}\ \bar{x} \to e_1 \text{ else } e_2 : \tau}\;\text{(T12)}$$

$$\frac{S; \Gamma; X \vdash_p v_q : \tau \qquad S; \Gamma \vdash \tau :: \star}{S; \Gamma; X \vdash_p v_q : \{x{:}\tau \mid x = v_q\}}\;\text{(T13)} \qquad \frac{S; \Gamma; X \vdash_q e : \tau' \qquad S; \Gamma \vdash \tau' <: \tau}{S; \Gamma; X, X' \vdash_p e : \tau}\;\text{(T14)}$$

The typing judgment $S; \Gamma; X \vdash_p e : \tau$ above states that an expression e, when typed with the privilege of principal p in an environment Γ and signature S, has type τ. The set X records a subset of the variables in Γ, and each element of X represents a capability to use an assumption in Γ. The rule (T1) requires data constructors to be used only in code granted the appropriate privilege. In the second premise of (T12), we type check a pattern $D\ \bar{\tau}\ \bar{x}$ to ensure that data constructors are also destructed in a context with the appropriate privilege.

In (T2) we type a non-affine variable x by looking up its type in the environment and checking that the privilege of the context matches that of the variable. (T3) is similar, but additionally allows an affine variable to be used only when a capability for its use appears in X. Unlike linear typing, affine assumptions need not always be used. (T4) allows an arbitrary number of assumptions X' to be forgotten, and for e to be checked with a privilege q that is not greater than privilege p that it has been granted. An expression is granted privilege by enclosing it in angle brackets, as shown in (T11).

Returning to the second premise of (K7), we check a type-level term v_p with the privilege of $\top$. The intuition is that in well-typed programs, type-level terms have no operational significance and, as such, cannot violate information-hiding. We also check v_p in (K7) with an empty set of capabilities X. According to the well-formedness rule of kind $\tau \rightarrow \kappa$, no well-formed type constructors can be applied to an affine value, so a type-level term like v_p never uses an affine assumption.

In (T5), we require fixed variables f to be given a non-affine type, and for the recursive expression to not capture any affine assumptions. In (T6), we check that the type of the formal parameter is well-formed, and type check the body in an extended context. We record the privilege p of the program point at which the variable x was introduced to ensure that x is not destructed in unprivileged code in the function-body e. In the conclusion of (T6), we use the auxiliary function $Q(X, \tau)$, which attaches an affine qualifier to τ if the function captures any affine assumptions from its environment. (T7) is similar. Typing let-expressions is standard, with the addition that the second premise of (T8) ensures that the let-bound variable x does not escape its scope in the type τ_2. When typing an application $v_q\ v_r$ in (T9), we split the affine assumptions among the sub-terms. We allow v_q to be a possibly affine function type—the shorthand $?\tau$ captures this, and we use the same notation in (T10).

We illustrate pattern-matching using an example from FileRM. Consider matching a value v_q of type tracked string (F file) against a pattern L string $x\ y$. When checking the true-branch, we record several term equalities that capture the runtime behavior of pattern matching. These assumptions will be used by our theorem prover in discharging proofs of refinement formulas (via the type conversion relation, discussed shortly). In our example, one such equality assumption is, clearly, $v_q \doteq$ L string $x\ y$. However, with FINE's value-indexed types, we can also infer equalities for some of the pattern-bound variables. In particular, by unifying the type of the scrutinee, tracked string (F file), with the type of the pattern, tracked string y, we can infer $y \doteq$ F file.

In (T12), we split the affine assumptions between v_q and the branches. In the second premise, we type the pattern and in the third premise, unify the type of the scrutinee with the type of the pattern to compute equalities among the term indices—the definition of the unification judgment is standard and we omit it from our presentation. The fourth premise checks e_1 with the computed equality assumptions. The last premise checks e_2 with no additional assumptions. A variation in which e_2 is checked with a disequality **forall** $\bar{x}.v_q \neq D\ \bar{\tau}\ \bar{x}$ is also feasible. However, in practice, we use n-way exhaustive pattern matching (**match** x **with** P1 $\rightarrow$ e1 ... Pn $\rightarrow$ en) and derive disequalities by relying on axioms that discriminate data constructors, e.g., **forall** $D_1, D_2, \bar{x_1}, \bar{x_2}, \bar{\tau}_1, \bar{\tau}_2. D_1 \neq D_2 \Rightarrow D_1 \bar{\tau}_1 \bar{x_1} \neq D_2 \bar{\tau}_2 \bar{x_2}$.

We use (T13) to give values a precise singleton type using an equality refinement. This is useful in bootstrapping the type conversion relation, used in the second premise of (T14), and defined below. Type conversion $S; \Gamma \vdash \tau <: \tau'$ is a reflexive, transitive relation without any structural rules, e.g., contra- and co-variant subtyping in function types. The type system of CIL uses nominal subtyping, and structural rules of this form are not easily translated. The rule (S3) is our interface to the solver—we discuss this with an example shortly. The rule (S4) treats a refined type as a subtype of the underlying type. Type conversion includes an equivalence relation on types $S;\Gamma \vdash \tau \cong \tau'$. In this judgment, (E5) allows a type-level term v_p to be equated with v'_p when an assumption $v_p \doteq v'_p$ appears in the context.

Type conversion: $S;\Gamma \vdash \tau <: \tau'$**,** $S;\Gamma \vdash \tau \cong \tau'$ **and** $S;\Gamma \vdash e \cong e'$
Where $S;\Gamma \models \phi$ is the first-order logic entailment relation

$$\frac{S;\Gamma \vdash \tau_1 \cong \tau_2}{S;\Gamma \vdash \tau_1 <: \tau_2}\text{(S1)} \quad \frac{\begin{array}{c}S;\Gamma \vdash \tau_1 <: \tau_2\\ S;\Gamma \vdash \tau_2 <: \tau_3\end{array}}{S;\Gamma \vdash \tau_1 <: \tau_3}\text{(S2)} \quad \frac{\begin{array}{c}S;\Gamma \vdash \tau <: \tau'\\ S;\Gamma, x{:}(p,\tau) \models \phi\end{array}}{S;\Gamma \vdash \tau <: \{x{:}\tau' \mid \phi\}}\text{(S3)}$$

$$\frac{}{S;\Gamma \vdash \{x{:}\tau \mid \phi\} <: \tau}\text{(S4)} \quad \frac{}{S;\Gamma \vdash \tau \cong \tau}\text{(E1)} \quad \frac{}{S;\Gamma \vdash v_p \cong v_p}\text{(E2)}$$

$$\frac{S;\Gamma \vdash \tau_1 \cong \tau'_1 \qquad S;\Gamma \vdash \tau_2 \cong \tau'_2}{S;\Gamma \vdash \tau_1\ \tau_2 \cong \tau'_1\ \tau'_2}\text{(E3)} \quad \frac{S;\Gamma \vdash \tau_1 \cong \tau'_1 \qquad S;\Gamma \vdash v_p \cong v'_p}{S;\Gamma \vdash \tau_1\ v_p \cong \tau'_1\ v'_p}\text{(E4)}$$

$$\frac{v_p \doteq v'_p \in \Gamma \lor v'_p \doteq v_p \in \Gamma}{S;\Gamma \vdash v_p \cong v'_p}\text{(E5)} \quad \frac{\forall i,j \qquad S;\Gamma \vdash \tau_i \cong \tau'_i \qquad S;\Gamma \vdash v_j \cong v'_j}{S;\Gamma \vdash D\ \bar{\tau}\ \bar{v} \cong D\ \bar{\tau}'\ \bar{v}'}\text{(E6)}$$

The key rule in type conversion related to refinement typing is (S3). This rule allows a type τ to be promoted to a refined type $\{x{:}\tau' \mid \phi\}$ when τ is a subtype of τ', and when our solver can deduce the formula ϕ from the typing context. The entailment relation $S;\Gamma \models \phi$ is standard—we illustrate its behavior using an example from §2.2. When typing the main loop of ConfWeb, we are required to construct a derivation of the form $S;\Gamma \vdash \mathrm{s} : \{\mathrm{x{:}st} \mid \mathrm{In\ a\ x}\}$, where (dropping principals for clarity) $\Gamma =$ s:st, a:attr, b:$\{$x:bool | x=true ⇒ In a s$\}$, b $\doteq$ true. We construct this derivation by using (T14) with (T13) in the first premise to derive $S;\Gamma \vdash s : \{\mathrm{x{:}st} \mid \mathrm{x{=}s}\}$, and a derivation of $S;\Gamma \vdash \{\mathrm{x{:}st} \mid \mathrm{x{=}s}\} <: \{\mathrm{x{:}st} \mid \mathrm{In\ a\ x}\}$ in the second premise. This latter derivation proceeds by using (S3), where we deduce $S;\Gamma, \mathrm{x}{:}\{\mathrm{x{:}st} \mid \mathrm{x{=}s}\} \models \mathrm{In\ a\ x}$ by using Z3 to show that the theory (s:st, a:attr, b:bool, b=true ⇒ In a s, b=true, x:st, x=s, not(In a x)) is unsatisfiable. Importantly, FINE's type system ensures that the theories we generate never contain any affine assumptions, thus eliminating the need for a linear logic prover.

3.3 Dynamic semantics

The operational semantics of FINE is instrumented to account for two program properties. First, our semantics places affinely typed values in a memory M. Reads from the memory are destructive—this allows us to prove that in well-typed programs, affine values are never used more than once. The semantics also tracks the privilege of expressions by propagating brackets through reductions, which is useful in showing an information-hiding property for our module system. The main judgment is written $(M, e) \overset{p}{\rightsquigarrow} (M', e')$, and states that given an initial memory M an expression e steps to e' and updates the memory to M'. The p-superscript indicates that e steps while using the privilege of the principal p. The omitted rules include reductions for let-bindings, standard beta-reduction for type and term applications, unrolling of fixed points, and pattern matching.

Dynamic semantics (selected rules)
Where a memory $M ::= (x, v_p), M \mid \cdot$

$$\langle v_p \rangle_p \overset{p}{\rightsquigarrow} v_p \ \text{(R1)} \qquad \langle\langle v_q \rangle_q\rangle_r \overset{p}{\rightsquigarrow} \langle v_q \rangle_q \ \text{(R2)} \qquad \langle \lambda x{:}\tau.e \rangle_q \overset{p}{\rightsquigarrow} \lambda y{:}\tau.\langle e[\langle y \rangle_p / x] \rangle_q \ \text{(R3)}$$

$$\frac{e \overset{q}{\rightsquigarrow} e'}{\langle e \rangle_q \overset{p}{\rightsquigarrow} \langle e' \rangle_q}\text{(R4)} \qquad \frac{\begin{array}{c} S;\cdot;\cdot \vdash v_p : \tau \qquad S;\cdot \vdash \tau :: \mathtt{A} \\ M' = M, (x, v_q) \qquad x \text{ fresh}\end{array}}{M, v_p \overset{p}{\rightsquigarrow} M', x}\text{(R5)} \qquad \frac{M = M_1, (x, v_q), M_2}{M, x \overset{p}{\rightsquigarrow} (M_1, M_2), v_q}\text{(R6)}$$

Reduction rules that do not involve reading from or writing to memory are written $e \overset{p}{\rightsquigarrow} e'$. All the interesting rules that manage privileges and brackets fall into this fragment. Redundant brackets around p-values can be removed using (R1). However, not all nested brackets can be removed, as (R2) shows. In (R3), a λ-binder is extruded from a function with q-privilege so that it can be applied to a p-value. We have to be careful to enclose occurrences of the bound variable in e within p-brackets, to ensure that e treats its argument abstractly. Finally, (R4) allows evaluation to proceed under a bracket $\langle\cdot\rangle_q$ with q-privilege. The rules (R5) and (R6) model memory operations. The rule (R5) is applicable non-deterministically. It allocates a new location x for an affine value v_p into the memory M and replaces v_p with x. When a location x is in destruct position, (R6) reads a value v_p from M and deletes x.

Theorem 1 establishes the soundness of FINE through the standard progress and preservation lemmas. In the statement below, all free variables are implicitly universally quantified. Additionally, we say that a memory M is typeable with an environment $S; \Gamma$, if $S; \cdot; \cdot \vdash_p M(x) : \Gamma(x)$, for each location $x \in \text{dom}(M)$. In addition to showing that well-typed programs do not go wrong, our soundness result guarantees that affine values are destructed at most once—a result that shows that state changes are modeled accurately. The proof appears in our technical report [20].

Theorem 1 (Soundness): *For all well-formed signatures S; environments Γ; non-values e; and memories M typeable with $S; \Gamma$, the following statements are true:*

1) *If $S; \Gamma; dom(M) \vdash_p e : \tau$ then there exists M', e' such that $M, e \overset{p}{\rightsquigarrow} M', e'$.*
2) *If $S; \Gamma; X \vdash_p e : \tau$ and $M, e \overset{p}{\rightsquigarrow} M', e'$ for some p, M', e', and $X \subseteq dom(M)$; then, there exists Γ', X' such that $S; \Gamma'; X' \vdash_p e' : \tau$ and M' is typeable with $S; \Gamma'$. Furthermore, for $\Delta_X = (dom(M) \cup dom(M')) \setminus (dom(M) \cap dom(M'))$ if $dom(M') \supseteq dom(M)$ then $X' = X \cup \Delta_X$; otherwise $X' = X \setminus \Delta_X$.*

3.4 Reasoning about the security of FINE programs

FINE allows programmers to specify conditions for correct policy enforcement and the type system checks that these conditions are satisfied. But, the onus is on the programmer to get these specifications right. For example, in the FileRM module of §2.1, wrongly assuming (**forall** p:prin. CanRead p f ⇒ CanRead p g) ⇒ CanFlow (F f) (F g) (instead of the Atomicflow assumption) would destroy any meaningful confidentiality property intended for FileRM to enforce. Similarly, in the Authentication module, forgetting to declare the cred type **private** would allow adversaries to forge credentials. In neither case would FINE's type checker complain. However, the metatheory of FINE provides a useful set of primitives using which an expert can prove high-level security properties. In prior work on the Fable calculus, we adopted a similar approach and showed how the metatheory of Fable could be used to prove high-level security properties (e.g., noninterference) for encodings of information flow, provenance tracking, and role-based access control. We anticipate a similar strategy being effective for FINE. Additionally, in §4, we discuss how tools like model checkers can complement FINE and be used to establish that FINE programs correctly enforce high-level security goals.

In addition to type soundness, the metatheory of FINE yields two general purpose security properties—proofs appear in our technical report. The first, corresponding to a secrecy property, is value abstraction. The theorem below states that a program e without p-privilege cannot distinguish p-values. As a corollary, we can also derive an integrity property, namely that a program without p-privilege cannot manufacture a p-value to influence the behavior of code with p-privilege. Note that this theorem appeals only to the pure fragment of our reduction rules—affine typing plays no special role in value abstraction. Additionally, observe that this result applies to selective information sharing/hiding between multiple principals, as FINE's module system includes a lattice of principals ordered by the $p \sqsubseteq q$ relation. Finally, although this theorem applies to the abstraction of a single value from the p module exported at type τ, the program e can contain code from several principals.

Theorem 2 (Value abstraction): *For well-formed signatures S and non-values e, if e uses a p-value x but is well-typed without p-privilege, (i.e., $S; x{:}(p,\tau); x \vdash_q e : \tau'$ and $p \sqsubseteq q \notin S$) and, except for $\langle x\rangle_p$, e is free of r-brackets $\langle\cdot\rangle_r$, for any r where $p \sqsubseteq r \in S$; then, for any pair of τ-typed values v_p^1 and v_p^2, (i.e., $S;\cdot;\cdot \vdash_p v_p^i : \tau$, $i \in \{1,2\}$) such that $e[v_p^1/x] \overset{q}{\rightsquigarrow} e_1$, there exists e' such that $e_1 = e'[v_p^1/x]$ and $e[v_p^2/x] \overset{q}{\rightsquigarrow} e'[v_p^2/x]$.*

4 Compiler implementation and application experience

We have implemented a prototype compiler, currently approximately 20,000 lines of F# code extending a front-end and IL generation libraries derived from the F# compiler [23]. The type-preserving translation of FINE to CIL accounts for a significant fraction of the complexity. Our compiler currently generates .NET assemblies that allow FINE programs to easily interface with modules defined in F#. Interoperability with the rest of .NET allows us to write only security critical parts of an application in FINE, leaving the rest to other, more commonly used languages.

The table below shows several small reference monitors in FINE, their size, the number of proof obligations generated during type checking, and parsing and type checking time (on 3.2 GHz Pentium Core Duo desktop running Windows Vista). Most benchmarks contain dense security critical code, where nearly every function call demands proving refinement formulas. Our results show that using an external solver to discharge these proofs (as opposed to constructing them by hand as in Fable or Aura) is critical for practical programming. We expect the checking time to improve significantly as we move from naïve representations of typing environments (currently association lists) to more efficient data structures.

Name	LOC	# pf. obl.	parsing/type checker time (s)
AuthAC	34	1	0.28
FileRM	120	36	1.64
FileAutomaton	121	3	0.45
IFlow	127	22	0.84
HealthWeb	318	19	6.41
DynDKAL	336	34	1.26
Lookout	519	23	2.73
ConfRM and ConfWeb	647	57	4.01
ProofLib	9943	0	19.28
Total	2222 (+ 9943)	195	17.62 (+ 19.28)

4.1 Modeling CONTINUE

Our most substantial example is the modeling of the security policy of CONTINUE. CONTINUE's authors provided us with a specification of its policy, partly in natural language and partly as specification in the Alloy modeling language [12]. Starting from this specification, we implemented ConfRM to enforce a policy that contains 9 phases and 12 actions. Policy assumptions in ConfPolicy describe when each action is permissible, and a function exposed in the external interface of ConfRM (with a suitable refinement type on the state) mediates access to this action. In addition, each action corresponds to a particular web request handled by ConfWeb.

A significant fragment of the Alloy specification for CONTINUE is devoted to specifying validity conditions on the authorization state. For example, in any given state, validity requires the assignment of papers to reviewers to respect the conflict of interest constraints. We found it relatively straightforward to express several of these validity constraints, although our implementation has yet to cover all the features of CONTINUE's specification. One simplifying assumption we make is that there is a unique phase for the entire conference. In contrast, the Alloy specification associates a phase with each paper, and different papers can be in different phases at any given time. Extending our attr type to account for this complexity is possible, though we have yet to implement this.

Our experience with CONTINUE illustrates an important aspect of FINE. Tools like Alloy are useful for reasoning abstractly about policies and establishing that these correctly specify high-level security goals. However, the abstract analysis of policies in Alloy is disconnected from system implementations that are expected to enforce these policies. FINE, in contrast, does not attempt to validate policies, but provides assurance that system implementations properly enforce their policy specifications. We view these two approaches as complementary and expect their combination to be a potent tool for security analysis of system implementations. For example, the Alloy specification includes assertions to check that no sequence of actions allows a principal to read or write a review when there is a conflict of interest. We plan to investigate using the metatheory of FINE and the types of ConfRM, in conjunction with a tool like Alloy, to prove such facts of our implementation.

4.2 Other benchmarks

The benchmark FileRM extends the example from §2.1 to account for confidentiality and integrity concerns when tracking information flow. Recall that in FileRM the lattice of security labels was derived from a specification of access control permissions using the Atomicflow assumption. To type check FileRM using Z3, we needed to rewrite the AtomicFlow assumption to the form shown below. To reason about formulas that use nested quantifiers, Z3 relies on a pattern-based instantiation mechanism that requires all bound variables (p in Atomicflow) to be guarded by non-equality predicates. Note that this is not a fundamental limitation of FINE. We are currently investigating the use of first-order solvers to reason directly about quantified formulas without this restriction. For example, a customized version of Coq's `firstorder` tactic can discharge proofs of the CanFlow proposition using the assumption AtomicFlow as shown in §2.1.

```
assume CW:IsPrin Admin && IsPrin (U ''Alice'') && IsPrin (U ''Bob'') && ...
assume AtomicFlow: forall f:file, g:file.
   (forall p:prin. (IsPrin p && CanRead p g) ⇒ CanRead p f) ⇒ CanFlow (F f) (F g)
```

Of the other benchmarks, AuthAC is a small purely permission-based access control monitor for files combined with password-based authentication. FileAutomaton is a reference monitor that implements an automaton-like policy on files, where, through the use of dependent and affine types, a file handle is indexed with a value indicating its current state, e.g., Open, Closed etc. A similar idiom could be used in ConfRM to associate phases with papers, instead of a global phase for the entire conference. IFlow is an implementation of a traditional information flow policy using a three-point lattice of labels which does not require the nested quantifiers of FileRM. HealthWeb is a reference monitor for an application that manages a database of electronic medical records. It enforces a stateful authorization policy. DynDKAL is an interpreter for an authorization logic; it uses refinement types to ensure that instantiations of quantified assumptions in policies is performed correctly. Lookout is the core reference monitor of a plugin-based email client we have started to build. This program mixes stateful authorization in the style of ConfRM with information flow tracking in the style of FileRM.

Finally, ProofLib is an automatically generated program, our largest test case by far. This program makes no use of refinement types and is used as a utility by our type-preserving compiler to represent proof terms. We include it here to give the reader a sense of the cost of dependent type checking for larger programs.

5 Related work and conclusions

Several programming languages and proof assistants use dependent types, including Agda [17], Coq [2], and Epigram [16]. All of these systems can be used to verify full functional correctness of programs. However, to ensure logical consistency of the type system, these languages exclude arbitrary recursion, making them less applicable for general-purpose programming. Projects like YNot [4] and Guru [19] aim to mix effects like non-termination with dependently typed functional programming; YNot also supports programming with state in an imperative style. Restrictions in both languages ensure that proofs are pure, ensuring that logical consistency is preserved. All of these systems include automation and tactic languages, but programmers still need to construct interactive proofs for their code. In contrast, FINE targets weaker, security properties; forgoes logical consistency in favor of practical programming by including recursion; and automatically synthesizes proof terms using an SMT solver. FINE also provides affine types to allow the enforcement of state-modifying policies, which could be expressed in YNot, but not easily in the other languages.

Dependent types have also been used for security verification. Jif [5] uses a limited form of dependent typing to express dynamic information flow policies. Aura [13] is specialized for the enforcement of policies specified in a policy language based on an intuitionistic modal logic. This makes Aura less applicable to policies specified in other logics, e.g., the Datalog-based policy language of Dougherty et al. [7], and Aura cannot model stateful policies. Aura provides logical consistency by separating types from propositions and excluding arbitrary recursion in proof terms. However, proof terms in Aura are always programmer-provided. As such, Aura is positioned as an intermediate language, rather than a source-level language. Fable [21], is another intermediate language for security verification that uses dependent types. Fable uses a two-principal module system. FINE's module system generalizes Fable's, with support for a lattice of multiple principals. FINE is also related to λAIR [22], a calculus that targets the enforcement of declassification policies. λAIR's combination of affine and dependent types does not lend itself to integration with a solver and it was never implemented.

Refinement types in FINE are related to a similar construct in RCF [1]. Refinement formulas in RCF are drawn from an unsorted logic, rather than using dependent-type constructors, as we do. The lack of dependent type constructors in RCF makes it difficult to derive typeable proof terms, crucial to our goal of a type-preserving compiler for FINE. Additionally, without dependent type constructors, it appears impossible to enforce information flow policies in RCF, although RCF's implementation, F7, does include dependent type constructors. RCF also lacks support for stateful authorization policies, although recent work shows how stateful policies can be modeled in F7 using a refined state monad [3]. However, the soundness of this encoding relies on a trusted compilation of the program in a linear, store-passing style. FINE's type system also allows the use of refined state monads, but, as discussed in §2.3, affine types in FINE also admit other stateful programming idioms.

FINE is also related to hybrid-typed languages that use refinement types, like Sage [9]. Sage uses a trusted external solver to discharge proofs; we extract typeable proof terms from Z3 rather than trusting it. Another difference is that Sage automatically insert runtime checks when the solver fails to discharge a proof obligation. Failed runtime checks can cause subtle leaks of information, so automatic insertion of runtime checks is not yet a feature of FINE, where security is the primary concern—we plan to investigate adding support for automatic policy checking in the future.

Conclusions. This paper has presented FINE, a language for enforcing rich, stateful authorization and information flow policies. Our experience constructing several reference monitors provides initial evidence that programming in FINE is practical, due in part to the use of an automated solver to ease the proof burden, and that FINE can be used to check the enforcement of security policies commonly applied to software.

Acknowledgments. Thanks to Shriram Krishnamurthi for providing us with a specification of CONTINUE's policy; to Nikolaj Bjørner and Leonardo de Moura for help with Z3; to Karthik Bhargavan, Johannes Borgstroem, Cédric Fournet, and Andy Gordon for numerous discussions about this work.

References

1. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.
2. Y. Bertot and P. Castéran. *Coq'Art: Interactive Theorem Proving and Program Development.* Springer Verlag, 2004.
3. J. Borgstroem, A. Gordon, and R. Pucella. Roles, stacks, histories: A triple for hoare. Technical Report MSR-TR-2009-97, Microsoft Research, 2009.
4. A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, 2009.
5. S. Chong, A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java + information flow, July 2006. Software release.
6. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
7. D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *LNCS*, 2006.
8. ECMA. Standard ECMA-335: Common language infrastructure, 2006.
9. C. Flanagan. Hybrid type checking. In *POPL*. ACM, 2006.
10. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*. ACM, 1993.
11. D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM TOPLAS*, 22(6), 2000.
12. D. Jackson. Alloy: a lightweight object modelling notation. *TOSEM*, 11(2), 2002.
13. L. Jia, J. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *ICFP*, 2008.
14. S. Krishnamurthi, P. W. Hopkins, J. Mccarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the PLT Scheme web server. *HOSC*, 20(4), 2007.
15. H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, 1984.
16. C. McBride and J. McKinna. The view from the left. *JFP*, 14(1), 2004.
17. U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers Institute of Technology, 2007.
18. V. Simonet. FlowCaml in a nutshell. In G. Hutton, editor, *APPSEM-II*, pages 152–165, 2003.
19. A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified programming in Guru. In *PLPV*, 2008.
20. N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. Technical Report MSR-TR-2009-164, Microsoft Research, 2009.
21. N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *S&P*, 2008.
22. N. Swamy and M. Hicks. Verified enforcement of stateful information release policies. In *PLAS*, 2008.
23. D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.
24. P. Wadler. Linear types can change the world. In *Prog. Concepts and Methods*, 1990.
25. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.