Type-preserving Compilation for End-to-end Verification of Security Enforcement

Juan Chen
Microsoft Research
juanchen@microsoft.com

Ravi Chugh
University of California, San Diego
rchugh@cs.ucsd.edu

Nikhil Swamy
Microsoft Research
nswamy@microsoft.com

Abstract

A number of programming languages use rich type systems to verify security properties of code. Some of these languages are meant for source programming, but programs written in these languages are compiled without explicit security proofs, limiting their utility in settings where proofs are necessary, e.g., proof-carrying authorization. Others languages do include explicit proofs, but these are generally lambda calculi not intended for source programming, that must be further compiled to an executable form. A language suitable for source programming backed by a compiler that enables end-to-end verification is missing.

In this paper, we present a type-preserving compiler that translates programs written in FINE, a source-level functional language with dependent refinements and affine types, to DCIL, a new extension of the .NET Common Intermediate Language. FINE is type checked using an external SMT solver to reduce the proof burden on source programmers. We extract explicit LCF-style proof terms from the solver and carry these proof terms in the compilation to DCIL, thereby removing the solver from the trusted computing base. Explicit proofs enable DCIL to be used in a number of important scenarios, including the verification of mobile code, proof-carrying authorization, and evidence-based auditing. We report on our experience using FINE to build reference monitors for several applications, ranging from a plugin-based email client to a conference management server.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Security, Verification, Languages, Theory

Keywords Security type systems, dependent types, bytecode languages, functional programming, authorization, information flow, mobile code security, compilers

1. Introduction

On today's internet, users concerned about their security and privacy would be well advised to be wary of the code they download and run on their computers. However, for the lack of an alternative, users routinely download complex programs (often as JavaScript in browsers, but also Flash, Java, and .NET plugins, applications for mobile phones, etc.) from unknown parties and allow these programs free access to their sensitive data. With the advent of cloud services, using the technologies currently at our disposal, users may also have no choice but to implicitly trust that service providers protect their data and computations properly.

As a step towards improving this state of affairs, we want users to be able to specify rich policies to control their security and privacy, and to receive formal proofs that the code they download, or the cloud services they rely upon, always respect these policies. But, the policies used in practice are complex, and properly enforcing them, let alone producing proofs, is known to be hard.

In response to this challenge, researchers have proposed several programming languages with rich type systems tailored towards proving security properties of code. However, a language with the ingredients to enable programmers to state and enforce complex real-world security policies—policies that mix aspects of authentication with stateful authorization and information flow controls—combined with a compiler that produces proofs that these policies are properly enforced does not yet exist.

For example, a number of prior languages, including Fable (Swamy et al. 2008), Aura (Jia et al. 2008), and PCML5 (Avijit et al. 2010) use *dependent types* to specify and enforce many kinds of policies, including (stateless) authorization and information flow controls. While type checking ensures that programs written in these languages are secure, programmers are required to construct proof terms to convince the type checker to accept their programs. The additional burden of programming with proofs causes the authors of at least Fable and Aura to position their systems as intermediate languages, rather than for source programming. Furthermore, all these languages are based on lambda calculi that must be compiled further to be executable on commodity systems.

In an effort to make source programming easier, languages like F7 (Bengtson et al. 2008) and FINE (Swamy et al. 2010) rely on theorem provers to automatically discharge proof obligations during type checking. These languages have been shown to be effective in verifying implementations of cryptographic protocols (F7) as well as checking that programs correctly enforce stateful authorization and information flow policies (FINE). However, to date, compilers for F7 and FINE do not produce verifiable security proofs, making them unsuitable for scenarios where proofs are needed, e.g., in mobile code settings where users would like to verify binaries; when security proofs need to be communicated between agents (as in proof-carrying authorization (Appel and Felten 1999)); or, when service providers need to construct audit trails for accountability (as in evidence-based audit (Vaughan et al. 2008)).

This paper presents a compiler that aims to fill this gap. Our compiler translates FINE programs to DCIL, an extension of the .NET Common Intermediate Language (CIL) (ECMA 2006), and affords source programmers the benefit of a reduced proof burden by automatically discharging proof obligations using the Z3 SMT solver (de Moura and Bjorner 2008). By extracting typeable proof terms from Z3, we also gain the benefits of compiling programs with explicit proofs. Additionally, by preserving types to the byte-

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

code level, we remove both the solver and much of our compiler from the trusted computing base (TCB), relying only on the DCIL type checker for bytecode verification and on the .NET virtual machine to faithfully execute the program. As such, we view our work as a stepping stone to future work that reduces the TCB further by, say, compiling DCIL to a typed assembly language (Morrisett et al. 1999). Meanwhile, our approach makes it possible to run FINE programs on stock .NET virtual machines; to interoperate with other more mainstream .NET languages; and to benefit from the libraries and tool support that come with the .NET platform.

1.1 Contributions

Theory. This paper makes three theoretical contributions. First, we formalize DCIL, an object-oriented bytecode language with dependent and affine types and prove it sound. Our extensions are designed to be small and compatible with the existing standard for CIL. Second, we formalize a translation from FINE to DCIL and prove that our translation preserves types. A final theoretical contribution is a source-to-source transformation of FINE programs, also proved to preserve types. We dub this transformation *derefinement* and explain its significance shortly.

A compiler implementation. We have implemented a compiler that translates FINE programs to executable CIL assemblies, verifiable using a type checker for DCIL. While this compiler remains under active development, an initial release is available on the web. A key component of our compiler is a module that translates proofs produced by Z3 into typeable proof terms in FINE. Type-checking proof terms produced by SMT solvers is an area of active research—we are aware of one project (Böhme 2009), concurrent with ours, that aims to reconstruct and check proofs for Z3.

Mobile-code security. Our prior work (Swamy et al. 2010) shows how FINE can be used to build secure reference monitors for server programs. In this paper, we show how FINE can be used in a mobile-code setting. Our main example shows how to use FINE to implement a reference monitor for a model of a plugin-based office utilities client—we call this model application LOOKOUT. Subject to a user's security policy, plugins can read emails in a user's inbox, make appointments in a calendar, send email responses, store data in a cookie store and selectively share this data with other plugins. We give examples of several kinds of security policies applied to plugins, including those that track information flows combined with role- and history-based authorization. Plugins for LOOKOUT can be shipped as DCIL assemblies and verified against a security policy before being installed by a user.

Experimental evaluation. We report on experiments using our compiler on about 12,000 lines of code, of which nearly 2,000 lines are from application programs or their reference monitors, and the rest a library of verified lemmas that simplify proof term construction. Despite further opportunities for optimization, the type checker of DCIL is already quite fast—it takes less than seven seconds to typecheck a 50MB assembly. However, compiling with proofs does impose an overhead—.NET assemblies that carry proof terms extracted from Z3 can, in some cases, be as much as 50 times larger than those without proofs. This is perhaps indicative of a bias in the SMT solving community to optimize for speed rather than for conciseness of proofs. Indeed, Z3 is among the few solvers that produce proof certificates at all. Our experiments include results from a simple custom first-order logic prover which, while not nearly as full-featured as Z3, is optimized to produce small proofs. When using this solver, we find that the overhead in code size due to proofs can be reduced to a factor of 2, i.e., a 25x improvement over Z3. This suggests that while the move towards certifying SMT solvers is a step in the right direction, there is much room for improvement in the size of proofs produced by these solvers.

1.2 Overview

Before we begin with the main content, we present a brief and informal overview of programming in FINE, and the main ideas behind the type-preserving compilation of FINE to DCIL.

Programming in FINE ***(§2).*** FINE is a functional programming language with a type system based on dependent refinement types. FINE also includes affine (use at most once) types to model state. While FINE can be used for general-purpose programming, we intend, primarily, for FINE to be used in the implementation of the security critical components of an application, e.g., a reference monitor. Programmers can give types that include security constraints to the sensitive resources in a program, and code that exposes an interface to these resources to the rest of the application can be type checked to ensure that it interposes the correct security checks.

For example, when enforcing an access control policy on files, a programmer may give the fread system call the following type: p:prin → cred p → {f:file | CanRead p f} → string. This is the type of a dependent function whose first parameter of type prin stands for the name of a principal in the system. As is usual with dependent types, the parameter can be given a formal name (here, p) which is bound to the right of the arrow. The next argument has the type cred p—this is a type indexed by a value (p) and stands for a credential authenticating the principal p. The third parameter is given a refined type {f:file | CanRead p f}, the type of all files f for which the proposition CanRead p f is true, i.e., those files that p is authorized to read. A security policy in FINE is defined using assumptions that grant privileges to some principals but not others. For example, the assumption **assume** CanRead Alice (File ''~/a.txt'') grants the read privilege on a specific file to the principal Alice. A reference monitor that provides mediated file access to an application is type checked against the declared type of fread and the policy assumptions. When this code attempts to give a file v:file a refined type {f:file | CanRead p f}, the FINE type checker verifies (using an external solver) that the refinement formula CanRead p v is deducible from the policy assumptions and from information about v available from the program context. Thus, the type checker ensures that every call to fread is mediated by the appropriate security checks.

Derefining FINE ***programs (§3).*** The first phase of our compilation chain is a source-to-source transformation called *derefinement*, in which values with refined types are associated with proof terms witnessing the deducibility of refinement formulas. For example, after derefinement, the type of fread becomes p:prin → cred p → f:file → proof (CanRead p f) → string. Our proof terms are in the style of LCF (Milner 1979), i.e., these are built using the constructors of an abstract data type proof t, where the constructors form a small trusted kernel that axiomatizes the inference rules of a first-order logic. Our compiler automatically constructs these proof terms by inspecting a natural-deduction style proof trace emitted by Z3.

Translating FINE ***to*** DCIL ***(§4).*** DCIL is an object-oriented bytecode language, designed to be a small extension to the type system of a side-effect-free fragment of CIL, where the additional type information used by DCIL is represented using the standard metadata facilities of .NET. The essence of DCIL lies in the way dependent function types are represented. Dependent functions in DCIL are instances of an abstract class DepArrow$<\alpha::\star, \beta::\alpha \to \star>$. This is a class with two parameters. The first parameter, α, is a type parameter of kind $\star$ (the kind for normal types) and is represented using standard CIL generics. The second parameter β uses a DCIL-specific feature—classes in DCIL can be parameterized by *type-level functions* and we use these to capture the functional dependences characteristic of dependent types. In this case, the second parameter of DepArrow is a function that constructs a $\star$-kinded type from an α-typed value. Turning to

our example, the derefined type of fread is translated (in part) to DepArrow<Prin, \p:Prin.DepArrow<Cred<p>, ...>>. Here, Prin is a class standing for the translation of the prin type in FINE; \p:Prin.DepArrow<Cred<p>, ...> is a type-level function from Prin-typed values to the DepArrow type. The Cred<p> type in the body of the function is a class *parameterized by a value*, another feature of DCIL, which we use to represent source-level value-indexed types like cred p. The translation of proof terms is no different from the translation of other program expressions. But, by including proof terms in the translation, the DCIL type checker is able to verify programs without the assistance of an external solver.

The extended version of this paper. We include in an extended version of this paper (Swamy et al. 2009) complete formalizations of the static and dynamic semantics of DCIL, the translation from FINE to DCIL, the derefinement translation, and proofs of the theorems in this paper. One important aspect of our full formalization is the special attention we pay to translating FINE's module system to DCIL, using CIL's access qualifier mechanisms. We prove that an information hiding property provided by FINE is preserved in the translation to DCIL—we make no further mention of this result in this paper. We also include a detailed description of some additional features of the LOOKOUT example—in particular, a treatment of information flow tracking in plugin code. The extended paper, a preliminary release of our compiler, and several example programs are available on the web at `http://research.microsoft.com/fine`.

2. FINE for mobile-code security

We begin by illustrating how FINE can be used to construct secure reference monitors and mobile code modules. We present fragments from LOOKOUT, a model, plugin-based office utilities client. The reference monitor for LOOKOUT mediates access to resources such as emails in a user's inbox, and is configured by a user-provided security policy that defines various access privileges. Our type-preserving compiler provides a number of benefits. First, by type checking the DCIL modules that represent LOOKOUT's reference monitor, an end-user receives assurance that it properly enforces her policy. Additionally, a user can download third-party plugins (as DCIL assemblies) that extend the core functionality of LOOKOUT. These plugins can be verified against the user's security policy before installation. Finally, LOOKOUT provides facilities to allow plugins to define policies to selectively share their data with other plugins—the types provide assurance to plugin developers that a plugin's private data is properly protected.

This section also aims to provide an introduction to programming in FINE. For a gentler and more thorough presentation of FINE, we refer the reader to our prior work (Swamy et al. 2010).

2.1 A reference monitor for LOOKOUT

Security objectives. LOOKOUT provides constructs for a user to specify a stateful role- and history-based authorization policy. A user can organize her contacts into roles, granting privileges to some principals but not others. The stateful aspects allow a user to change role memberships dynamically. Additionally, the reference monitor also records events like the sending of emails. A user can define a history-based policy over these events to, for example, ensure that plugins never spam a user's contacts by responding to emails repeatedly. Our implementation augments the fragment shown here with a number of additional features, including selectively sharing cookies between plugins using a plugin-provided access control policy, and information flow tracking through plugin and reference monitor code. We discuss these elements in §5.

Figure 1 shows a fragment of the API exposed by the LOOKOUT reference monitor to plugins. FINE uses a syntax based loosely on F# (Syme et al. 2007)—we point out differences along the way.

```
1  module LookoutRM
2  type prin (* the type of principals *)
3  private type cred :: prin → ⋆ (* cred p is a credential authenticating p *)
4  private type email = {sender:prin; contents:string}
5  val mk_email: prin → string → email
6  val sender: e:email → {p:prin | p=e.sender}
7  type evname = MsgIn:evname | MsgOut:evname | ...
8  type event :: evname → ⋆ → ⋆ = Event : e:evname → α → event e α
9  (* A vocabulary for an authorization policy *)
10 type action = ReadEmail: email → action
11             | ReplyTo: email → action
12             | SetCookie: prin → string → action
13             | ReadCookie: prin → string → action
14 type perm = Permit: prin → action → perm
15 type role = User:role | Friend:role | Plugin:role | ...
16 type att = Role: prin → role → att
17          | HasRepliedTo: prin → email → att
18 type st = list att
19 (* An affinely typed revokable signature of the program state *)
20 private type StateIs :: st → A = Sign : s:st → StateIs s
21 (* Propositions to define authorization constraints *)
22 type In :: att → st → ⋆ (* list membership *)
23 type Derivable :: perm → st → ⋆ (* dynamically derived perms *)
24 (* Useful type abbreviations *)
25 type ok<p:prin, a:action> = {s:st | Derivable (Permit p a) s}
26 type plus<s:st, a:att> = {x:st | In a x && forall (b:att).In b s ⇒ In b x}
27 (* An API for plugins *)
28 val readEmail: p:prin → cred p → e:email →
29                s:ok<p, ReadEmail e> →
30                StateIs s → (string * StateIs s)
31 val replyTo: p:prin → cred p → e:email → s:ok<p, ReplyTo e> →
32              StateIs s → (s':plus<s,HasRepliedTo p e> * StateIs s')
33 val setCookie: p:prin → cred p → name:string → value:string →
34                s:ok<p, SetCookie p name> → StateIs s → StateIs s
35 val getCookie: p:prin → cred p → owner:prin → name:string →
36                s:ok<p, ReadCookie owner name> → StateIs s →
37                (option string * StateIs s)
```

Figure 1. A fragment of a reference monitor for LOOKOUT

The types given to this API specify authorization constraints; for example, looking ahead to the readEmail function on line 28, we see an argument s:ok<p, ReadEmail e> which represents a constraint that the principal p have the ReadEmail permission on the email e. In Section 2.2, we show how a user can configure the behavior of this reference monitor by specifying a policy to grant privileges to certain principals and not others. Section 2.2 also shows code for a plugin. In the rest of this section, we proceed through Figure 1 sequentially, describing each element in detail.

Line 2 shows the type of principals prin—its representation is irrelevant to the example. We could, for example, use strings standing for user names, or public keys. Line 3 shows a type constructor cred that we use to represent authentication credentials. The constructor cred is given the *kind* prin → ⋆. (Kind ascriptions are written using double colons, while type ascriptions use single colons.) The kind of cred indicates that it constructs a type of kind ⋆, from a *value* of type prin; in other words, cred is a dependent type constructor. As with principals, the concrete representation of credentials is irrelevant. However, to ensure that credentials cannot be forged, we tag the cred type with the **private** qualifier ensuring that values of type cred p cannot be constructed directly by code not trusted by the LookoutRM module. (In practice, rather than including prin and cred in the definition of LookoutRM, we use a library (trusted by LookoutRM) that implements various principal representations and authentication schemes.)

Line 4 shows the type email. Our intention is to allow the user to define policies to protect access to the contents of an email. By declaring email private, FINE's module system ensures that untrusted clients of LookoutRM cannot directly project fields from

the email record. However, unlike for the cred p type, we do not aim to ensure the authenticity of emails. So, at line 5 LookoutRM exposes a function mk_email to allow clients to construct an email. At line 6, we provide an accessor to examine the sender field of an email without restriction—access control will apply only to the contents field. The type of sender shows it to be a dependent function, where the formal parameter named e is in scope to the right of the arrow. The return type of sender uses a refinement type to specify that the value p returned is in fact the sender field of the formal parameter e. In general, refinement types in FINE have the form $\{x{:}\tau \mid \phi\}$, where x is the formal name of a value of type τ, and x is bound in ϕ, a type that represents a first-order formula (with equality) (§3).

The design of LOOKOUT is based on a model that allows plugins to subscribe to various events, e.g., email arrival, message composition, etc. Lines 7-8 show the type of event names evname and the type event n t, consisting of an event name n and some payload of type t generated when the event is triggered. In FINE, each constructor of a variant can construct a different type. Thus, unlike F#, the constructors of a variant are decorated with their complete type.

At lines 10-18 we define various types that form a vocabulary for a security policy. Permissions (the type perm) are of the form Permit p a, which means that the principal p has the privilege to perform action a. Actions (type action) include reading from emails and replying to emails, as well as getting and setting cookies. Cookies are identified by a pair of the principal p that owns the cookie, and the cookie's key represented as a string.

The type st shown at line 18 forms the basis of the stateful authorization policy implemented by LOOKOUT. We re-use a model for stateful authorization which we have previously proposed (Swamy et al. 2010), which in turn was based on a model by Dougherty et al. (2006). In this model, authorization policies are specified as inference rules that derive permissions from a set of basic authorization attributes, where the attributes can change over time. For example, the attributes may include assertions about a principal's role membership, and the policy may include inference rules that grant permissions to principals in certain roles. The type att (lines 16-17) defines the attributes used in our scenario. The currently active role memberships of a principal are of the form Role p r. The attribute HasRepliedTo p e is used to record a message-reply event. In practice, several other relations (e.g., event subscriptions) are maintained in the state st of attributes.

Line 20 uses *affine types* in FINE, a key feature that allows state changes to be modeled. Types in FINE are classified into two kinds: $\star$, the kind of normal types, and A, the kind of affine types. Values with affine types may be used at most once. The notation Statels :: st → A indicates that Statels constructs an affine type from an st value. A value v:Statels s is a signature from the reference monitor attesting that s:st holds the current authorization attributes.

Next, at lines 22-23, we show two propositions with which to state authorization constraints in types. (Unlike Coq (Bertot and Castéran 2004) or Aura (Jia et al. 2008), FINE's kind system does not distinguish types and propositions.) The proposition In (line 22) is the standard list membership proposition, specialized to the st type. The proposition Derivable p s states that the permission p is derivable from the authorization attributes in s. Lines 25-26 show convenient abbreviations that use these propositions to define refined types. The type ok<p,a> is a refinement of st to those values in which p has the permission to perform the action a. The type plus<s, a> is a st that extends s with the attribute a.

Finally, we show a few functions exposed by LookoutRM to its clients. All the functions require the caller p to authenticate itself by passing in a credential cred p. To read an email e using the readEmail function, the caller p must show that it holds the ReadEmail e privilege in the current authorization state s. The return

```
1  module UserPolicy
2  open LookoutRM
3  assume U1: forall (p:prin) (e:email) (s:st).
4    In (Role p Plugin) s && In (Role e.sender Friend) s ⇒
5    Derivable (Permit p (ReadEmail e)) s
6  assume U2: forall (p:prin) (e:email) (s:st).
7    In (Role p Plugin) s && not (In (HasRepliedTo p e) s) ⇒
8    Derivable (Permit p (ReplyTo e)) s
9  assume U3: forall (p:prin) (n:name) (s:st).In (Role p Plugin) s ⇒
10   Derivable (Permit p (SetCookie p n)) s &&
11   Derivable (Permit p (ReadCookie p n)) s
12 (* A plugin module *)
13 module ApptMakerPlugin
14 open LookoutRM
15 val detectAppt: prin → string → option ({key:string; value:string})
16 val me:prin
17 type pst<p:prin> = (s:{x:st | In (Role p Plugin) x} * Statels s)
18 val hdlMsg: cred me → event MsgIn email → pst<me> → pst<me>
19 let hdlMsg c (Event e email) (s, tok) =
20  let c1 = contains s (Role (sender email) Friend) in
21  let c2 = contains s (HasRepliedTo me email) in
22  if c1 && not c2 then
23   let (contents, tok) = read_email me c email s tok in
24   match detectAppt (sender email) contents with
25    | None → (s, tok) (* no appointment extracted; do nothing *)
26    | Some {key=k; value=v} →
27      let tok = setCookie me c k v s tok in
28       replyTo me c email (``Confirm appt: ``^ v) s tok
29  else (s,tok) (* can't read email, or already sent notification *)
```

Figure 2. A user's policy and fragment of plugin code

value of readEmail is a pair containing the contents of the email as a string, and a signature asserting that authorization state is unchanged. The type of replyTo is similar, except its return value is given a dependent pair type, (x:t * t'), where x names the value in the first component of the pair and is bound in the type t'. The dependent pair in replyTo shows an updated state of the program s' which extends s with an event HasRepliedTo p e, and a signature attesting that s' is the new authorization state. We use the affinity of Statels s to model state updates. Since the caller of replyTo has used a v:Statels s value by passing it as an argument to replyTo, the types ensure that the caller can no longer use v to claim that the old state s is valid. The setCookie function allows a plugin with appropriate privilege to write a cookie. The getCookie function allows a plugin p to retrieve a cookie owned by another plugin owner only if p holds the appropriate privilege—if a cookie by that name does not exist, getCookie returns None, the empty constructor of the option type.

2.2 A LOOKOUT user's policy and a plugin

Figure 2 shows a module UserPolicy that configures the behavior of the LookoutRM reference monitor with several formulas that represent user-provided policy assumptions. The policy shown here is particularly simple—we discuss the policy used with our real implementation in §5. Assumption U1 allows a plugin to only read emails from friends. U2 allows a plugin to reply to an email e only if a reply has not already been sent. U3 allows plugins to set and get only the cookies it owns.

The rest of Figure 2 shows fragments from a plugin program. At a high-level, this plugin responds to incoming messages (MsgIn events), scans the contents of these messages for text that appears to be an appointment, and maintains a calendar of appointments for the user in the cookie store. The plugin also sends a response notifying the message sender that an appointment has been created. The custom logic of the plugin is represented by the function detectAppt, whose implementation is not shown.

The plugin defines a distinguished principal me (representing the plugin itself) at line 16. In hdlMsg at lines 20-21, we use the

function contains: s:st→ a:att→ {b:bool | b=true ⇔ In a s} (a standard tail-recursive list membership test implemented in FINE, but not shown here) to check if it has the privilege to read the email and reply. If the check succeeds, hdlMsg reads the contents of the email. If it detects an appointment, it sets a cookie in the store for the appointment and sends a reply confirming the appointment.

2.3 Discussion

In subsequent sections, we present a translation that compiles FINE programs in a proof-carrying style to DCIL. Before we proceed to the formalism, we discuss several benefits (and limitations) in the design of LOOKOUT and of our compiler.

Loosely coupled policy and code. LOOKOUT's design enables a good separation of security policy from code. Users declare a policy using high-level logical rules; these rules are then connected to the code using types. For example, the ReadEmail privilege granted by U1 in Figure 2 also appears in the type of readEmail at line 28 of Figure 1. But, the conditions under which such privileges are granted are declared only in the policy, not in the code—different users may define different conditions under which the ReadEmail privilege is granted. In practice, rather than requiring a user to write down the UserPolicy module directly, we expect that it can be generated from some high-level specification or interface in which to declare security preferences. One limitation, however, is that although the policy appeals to dynamically changing attributes, policy assumptions are required at compile time to type check the program. In the future, we plan to explore designs in which the reference monitor is configured by a purely dynamic policy.

Compiling to DCIL ***enables .NET interoperability.*** Rather than expecting plugin authors to program entirely in FINE, our compiler generates code that can easily call, and be called from, other .NET languages. For example, in the ApptMakerPlugin of Figure 2, the detectAppt function which handles the plugin's custom logic, could be implemented in any .NET programming language. Our compiler produces code that tries, where possible, to mimic F#'s object representations, which makes it convenient to interoperate with F#, but we have also used C# and ASP.NET.

However, interoperating with other .NET languages does require some care. For one, allowing C# programs to directly call a FINE function that expects an argument with a value-indexed type (e.g., cred p) is unsound, since the standard .NET bytecode verifier does not check that the argument actually has a type with the required index. Similar issues arise with exposing affine types to the rest of .NET. To defend against this, we take care to ensure that the values exposed to other .NET languages do not have FINE-specific types. However, our compiler does not yet check this automatically.

Calling other .NET languages from FINE also requires some care. The LOOKOUT application, as shown, only aims to enforce an authorization policy. When enhancing LOOKOUT with, say, information flow controls, one must be careful if calls to external functions are permitted. For example, an implementation of detectAppt in F# can easily leak the contents of an email, both via implicit or explicit flows. Since proper enforcement of a noninterference-like property (Sabelfeld and Myers 2003) requires tracking flows throughout an application, if a policy includes information flow constraints, then most (if not all) of the application must be written in FINE and type checked for security.

Efficient client-side verification. By compiling plugins to DCIL, clients can verify the security of plugins before installation. As our experimental results (§5) show, although carrying explicit proofs in DCIL increases code size significantly, type-checking DCIL assemblies is fast—we view fast checking times as a key enabler for mobile code verification. As discussed in the Introduction, proof terms can also be useful at runtime in support of applications like proof-carrying authorization. However, in the case of LOOKOUT, proof terms have little utility beyond verification. In support of such scenarios, we plan to implement an erasure pass for DCIL that can erase computationally irrelevant proof terms after verification.

values	v	$::=$	$x \mid D\ \overline{\tau}\ \overline{v} \mid \lambda x{:}t.e \mid \Lambda\alpha{::}\kappa.e$
expressions	e	$::=$	$v \mid v_1\ v_2 \mid v\ \tau \mid \text{let } x = e_1 \text{ in } e_2$
		$\mid$	$\text{let } (x, y) = e_1 \text{ in } e_2$
		$\mid$	$\text{match } v \text{ with } D\ \overline{\tau}\ \overline{x} \rightarrow e_1 \text{ else } e_2$
types	τ, ϕ	$::=$	$\alpha \mid x{:}\tau_1 \rightarrow \tau_2 \mid (x{:}\tau_1 * \tau_2) \mid \forall\alpha{::}\kappa.\tau$
		$\mid$	$T \mid \tau_1\ \tau_2 \mid \tau\ v \mid\ !\tau \mid \{x{:}\tau \mid \phi\}$
kinds	κ	$::=$	$\star \mid \mathtt{A} \mid \kappa \rightarrow \kappa \mid \tau \rightarrow \kappa$
signature	S	$::=$	$T{::}\kappa \mid D{:}\tau \mid S, S' \mid \cdot$
type env.	Γ	$::=$	$\alpha{::}\kappa \mid x{:}\tau \mid v \doteq v' \mid \Gamma, \Gamma' \mid \cdot$

Figure 3. Core syntax of FINE

3. Derefining FINE

In this section, we present a core syntax for FINE and describe (using several examples) the key aspects of *derefinement*, an initial source-to-source translation implemented by our compiler. Derefinement provides a way to associate explicit proofs of refinement formulas with the values that inhabit refined types. The main subtlety in derefinement is formulating it in a manner consistent with FINE's subtyping relation on refinement types. We also discuss theorems that establish that derefinement is sound and complete.

3.1 Core syntax of FINE

We begin by presenting a core syntax for FINE, shown in Figure 3. We adopt an A-normal presentation (Flanagan et al. 1993) of FINE. This helps to simplify the translation, and is convenient for giving names to expressions that index types. FINE values v are variables, full applications of n-ary data constructors D, and value and type abstractions. The expression forms include application, type application, two forms of let-bindings (the second is used to unpack dependent pairs), and a pattern matching construct. The types τ include type variables, dependent functions, dependent pairs, quantified types, type constructors and their applications to types or values, types with affine qualifiers $!\tau$, and refinement types. Types are classified according to their kind κ, where $\star$ is the kind of unrestricted types, and $\mathtt{A}$ is the kind of affine types. An important aspect of FINE's kinding system is that dependent type constructors, types with kind $\tau \rightarrow \kappa$, are only well-formed when the type τ has kind $\star$—indexing types with affine values is prohibited. We have argued (Swamy et al. 2010) that that this restriction is key to discharging proofs obligations using off-the-shelf classical provers, rather than requiring linear logic provers—we find that this restriction also simplifies the construction of proof terms. Programs are translated in the presence of a signature S that assigns kinds and types to all type and data constructors; and a typing environment Γ, which, in addition to variable bindings, contains equality assumptions $v \doteq v'$ that record the results of pattern matching tests.

3.2 Representing refinement formulas and proofs

Formulas that appear in refinements and in assumptions are represented using the type language—we generally use the meta-variable ϕ for types that stand for formulas or proofs of formulas. The logical connectives in formulas are represented using type constructors, e.g., And::$\star \rightarrow \star \rightarrow \star$, Or::$\star \rightarrow \star \rightarrow \star$, Not::$\star \rightarrow \star$, and quantified formulas are represented using the binding constructs provided by dependent types. A universally quantified formula $\forall(x{:}\tau).Px$, is represented as a dependent function $x{:}\tau \rightarrow Px$, where $P{::}\tau \rightarrow \star$; existential quantification $\exists(x{:}\tau).Px$ is represented using a dependent pair $(x{:}\tau * Px)$.

$$\frac{S;\Gamma \vdash \tau \overset{\text{bare}}{\hookrightarrow} \tau' :: \kappa \quad S;\Gamma, x{:}\tau' \vdash \phi \overset{\text{bare}}{\hookrightarrow} \phi' :: \star}{S;\Gamma \vdash \{x{:}\tau \mid \phi\} \overset{\text{box}}{\hookrightarrow} (x{:}\tau' * \mathsf{proof}\ \phi') :: \kappa}\ \text{(D1)} \qquad \frac{S;\Gamma \vdash \tau_1 \overset{\text{bare}}{\hookrightarrow} \tau_1' :: \kappa \quad S;\Gamma, x{:}\tau_1' \vdash \tau_2 \overset{b}{\hookrightarrow} \tau_2' :: \kappa'}{S;\Gamma \vdash x{:}\tau_1 \to \tau_2 \overset{\text{bare}}{\hookrightarrow} x{:}\tau_1' \to \tau_2' :: \star}\ \text{(D2)} \qquad \frac{S;\Gamma \vdash \tau_1 \overset{\text{box}}{\hookrightarrow} (x{:}\tau_1' * \phi) :: \kappa \quad S;\Gamma, x{:}\tau_1', y{:}\phi \vdash \tau_2 \overset{b}{\hookrightarrow} \tau_2' :: \kappa'}{S;\Gamma \vdash x{:}\tau_1 \to \tau_2 \overset{\text{bare}}{\hookrightarrow} x{:}\tau_1' \to y{:}\phi \to \tau_2' :: \star}\ \text{(D3)}$$

Figure 4. Selected rules from the derefinement of FINE types: $S;\Gamma \vdash \tau \overset{b}{\hookrightarrow} \tau' :: K$, where $b ::= \text{bare} \mid \text{box}$

We use an LCF-style (Milner 1979) proof system. Values of an abstract datatype (ADT) proof::$\star \to \star$ represent proofs of formulas. The constructors of this ADT represent inference rules that axiomatize a classical first-order logic with equality. User-provided assumptions are treated as additional data constructors of the proof type—we give an example in Section 3.4. We show a few rules from the proof kernel below.

```
T: proof True
Destruct_false: proof False → proof α
And_elim_1: proof (And α β) → proof α
Iff_elim_1: proof (Iff α β) → proof (Imp α β)
Modus_ponens: proof α → proof (Imp α β) → proof β
Bind_pf: proof α → (α → proof β) → proof β
```

The design of FINE's proof kernel is influenced by the features and limitations of the type system of our target language. In designing DCIL, we aimed to produce a minimal extension of CIL, without requiring any changes to existing CIL features. One of the limitations of CIL is that parametric polymorphism is only permitted on types of kind $\star$ with no support for quantification over types with higher kinds, e.g., $\star \to \star$. This prevents us from using a higher-order logic to represent proof terms in FINE.

This restriction manifests itself primarily in our treatment of equality. In addition to the core inference rules, we generate proof axioms for a first-order treatment of equality for each type defined in the program. For example, for the att type defined in Figure 1, we automatically generate a type Eq_att corresponding to equality for att values, and substitution axioms relating Eq_att to other propositions in the program. Some of these auto-generated types and axioms are shown below.

```
type Eq_bool:: bool → bool → ⋆
Refl_bool: b:bool → proof (Eq_bool b b)
type Eq_att:: att → att → ⋆
Refl_att: a:att → proof (Eq_att a a)
Mono_In_1: a:att→ b:att→ proof (Eq_att a b)→
                s:st→ proof (In a s)→ proof (In b s)
```

Finally, although not shown here, FINE has support for arbitrary recursion, so we do not claim that our proof system is logically consistent. However, the soundness of FINE and DCIL's module systems guarantee that proof terms are constructed using only the data constructors from our proof system and the user-supplied axioms, and that if a proof term has a normal form, then that normal form has the desired type. We view recovering logical consistency in the presence of recursion as an orthogonal issue, addressed either by tracking non-termination as an effect; by separating types from propositions and excluding recursion in the propositional fragment (as in Aura); or, by adopting a more permissive approach such as Operational Type Theory (Stump et al. 2008).

3.3 The derefinement translation

The derefinement translation associates explicit proofs of formulas $\phi[v/x]$ with values v given refined types $\{x{:}\tau \mid \phi\}$. The standard approach to this problem is to translate refinement types $\{x{:}\tau \mid \phi\}$ to dependent pairs $(x{:}\tau * \mathsf{proof}\ \phi)$—for example, Coq (Bertot and Castéran 2004) adopts exactly this strategy to represent refinement types (sometimes called subset types, in Coq terminology).

However, the standard approach faces a difficulty in our context, because the FINE type system (unlike Coq's) comes equipped with a subtyping relation, according to which $\{x{:}\tau \mid \phi\} <: \tau$. Since the representation of a $(x{:}\tau * \mathsf{proof}\ \phi)$ value clearly differs from the representation of a τ value, a naïve translation is incompatible with FINE's subtyping relation. Altering FINE's subtyping relation is not an option since it is key to the usability of FINE as a source programming language. Treating $\{x{:}\tau \mid \phi\}$ as a subtype of τ allows programmers to simply use refinement formulas to state invariants of their objects, but to otherwise use these values normally, without needing to insert tedious operations to pack and unpack dependent pairs of values and their proofs. The novelty of derefinement, then, lies in the way it selectively introduces dependent pairs in a manner that allows us to accommodate FINE's subtyping relation.

Figure 4 shows a few key rules from our derefinement judgment. This judgment is written $S;\Gamma \vdash \tau \overset{b}{\hookrightarrow} \tau' :: \kappa$, and reads that in a context with a signature S and environment Γ (wherein all types have already been derefined), a source type τ is derefined to τ' of kind κ. The superscript b is one of two constants: "bare" or "box". In the latter case, this indicates that the type τ was translated to a dependent pair of the form $(x{:}\tau' * \mathsf{proof}\ \phi)$—values of this type are "boxed" with a proof of the formula ϕ.

The rule (D1) shows a refinement type translated to a pair. The rule is simplified by assuming that refinement types are not nested. It is always possible to normalize types to this form, e.g., $\{x{:}\{x{:}\tau \mid \phi_1\} \mid \phi_2\}$ can be normalized to $\{x{:}\tau \mid \phi_1 \wedge \phi_2\}$. In (D2), we show the translation of a function type, where the argument type τ_1 is translated to the (unboxed) type τ_1'. The interesting case is the translation of functions that receive arguments with refined types, shown in rule (D3). Here, the argument is first translated to a boxed type, but, in the conclusion, we use a curried representation of a dependent pair. The effect of this formulation is that refinement types that appear in negative position are translated in a curried style, while those that appear in positive position are translated to dependent pairs. This serves two purposes. First, in the body of a function with this type, the argument x can be used at the type τ' without needing any coercions. More importantly, the name x of type τ' is bound in the return type τ_2, where it may, for example, index another type. Failing to curry the dependent pair would requiring the coercing occurrences of x in τ_2 to project out the first component of the dependent pair. Since FINE and DCIL only support value indexed types, inserting such coercions at the type level is not feasible.

The derefinement of expressions has a similar form: $S;\Gamma;X \vdash e \overset{b}{\hookrightarrow} e' : \tau$, where the context X records the set of affine assumptions usable by e. We omit this judgment due to space constraints. Instead we illustrate its behavior on a (simplified) fragment of the example program from §2. The top of Figure 5 shows the derefinement of types in a context—note the distinction between the translation of refinements in positive and negative contexts in the type of contains and readFoo respectively. We use me, s, tok and p as free variables bound in the context throughout the rest of this section. The source program on the left gives the boolean b a refined type. We type the **then**-branch of the conditional with the assumption that $\mathsf{b} \doteq \mathsf{true}$. At the right we show the derefined program. Values that are given boxed types, like b', are unboxed immediately to bind

me:prin, s:st, tok:Statels s, p:perm contains : s:st → a:att → {b:bool | b=true ⇔ In a s} readFoo : s:{x:st | Derivable p x} → Statels s → Statels s	me:prin, s:st, tok:Statels s, p:perm contains : s:st → a:att → (b:bool ∗ proof(Iff (Eq_Bool b true) (In a s)))) readFoo : s:st→ proof (Derivable p s) → Statels s → Statels s
1. **let** f = contains s **in let** b = f (Role me Plugin) **in**	1. **let** f = contains s **in let** b' = f (Role me Plugin) **in**
	1.1. **let** (b, pf1) = b' **in** *(∗values packed with proofs are unpacked immediately ∗)*
2. **if** b **then let** g = readFoo s **in**	2. **if** b **then let** h = readFoo s **in**
	2.1. **let** g = h (*v*:proof (Derivable p s)) **in** *(∗v uses pf1, and assumption b=true ∗)*
3. g tok **else** tok	3. g tok **else** tok

Figure 5. A source program (left) and its derefined version—an auto-generated proof term v is shown with its type ascribed.

both the underlying value and the proof in the context. The call to the function h:proof(Derivable p s) → Statels s → Statels s requires a proof term as an argument. The auto-generated proof term is shown as the value v, shown enclosed in a box in the figure.

To discharge proofs, our compiler constructs a first-order theory for Z3 by collecting user-provided axioms, variable bindings and match assumptions from the type environment Γ, e.g., bindings for normal variable like b, proof terms like pf1, and, in the **then**-branch of the conditional, the assumption b $\doteq$ true. We then assert the negation of the goal, (e.g, not (Derivable p s)) in this theory. When successful, Z3 determines that the theory is unsatisfiable and produces a proof. We translate this proof into a FINE value of type proof(Derivable p s) using the constructors of our proof kernel.

3.4 Generating proof terms

Consider typing the program of Figure 5 in the presence of the user-provided assumption:
assume U: **forall**(s':st). In (Role me Plugin) s' ⇒ Derivable p s'. Note that p:perm is bound in the context. This assumption is translated to the data constructor U shown below. At the call to readFoo, we are required to construct a term with type proof(Derivable p s). We show such a term below (omitting type instantiations for clarity):

```
U: proof (s':st → proof (Imp (In (Role me Plugin) s') (Derivable p s')))
Bind_pf U
  λf:(s':st → proof (Imp (In (Role me Plugin) s') (Derivable p s'))).
    Modus_ponens (Modus_ponens ((Refl_bool b):proof(Eq_bool b true))
                               (Iff_elim_1 pf1))
                 (f s)
```

The proof proceeds by applying the monadic bind operator on the assumption U; then applying the quantified assumption in U to s, the state variable in question; and eliminating the implication proof (Imp (In (Role me Plugin) s) (Derivable p s)) using the Modus_ponens rule. To obtain a proof of In (Role me Plugin) s, we make use of the proof term pf1 that is introduced in the context at line 1.1 on the right side of Figure 5. Recall that pf1 has type proof (Iff (Eq_bool b true) (In (Role me Plugin) s)). We convert this to an implication proof (Imp (Eq_bool b true) (In (Role me Plugin) s)) by applying Iff_elim_1 to pf1. Finally, we eliminate this implication using an application of Modus_ponens, where we construct a proof of (Eq_bool b true) using the term (Refl_bool b):proof(Eq_bool b true). Notice, however, that the type of Refl_bool b is proof(Eq_bool b b). To check the type ascription, we rely on a match assumption in the context, b $\doteq$ true, introduced in the **then**-branch of the **if**-statement on line 2. Given this assumption, our type checker equates the type proof(Eq_bool b b) with proof(Eq_bool b true), as required by the context, and completing the proof. Note that the match assumptions that induce type equivalences require special treatment in the translation to DCIL—we discuss this in §4.4.

Our implementation uses Z3 to synthesize proof terms similar to (but often considerably larger than) the one shown above. Our approach to translating Z3 proofs is syntax-directed—most Z3 proof steps are processed locally and just once. The exceptions to this rule are proof steps that deal with equisatisfiable formulas, using principles such as skolemization. As with equality, a general treatment of these formulas requires a use of higher-order logic. Nevertheless, for the few cases in our benchmarks that require equisatisfiability, we are able to provide a suitable first-order treatment to extract typeable proof terms. In the future, we plan to explore adding limited forms of higher-order quantification to DCIL, while still being faithful to the restrictions imposed by CIL generics. We anticipate such a feature simplifying the construction of proof terms significantly.

We conclude this section by presenting Theorem 1 which establish the soundness and completeness of derefinement. In the statement below, $S;\Gamma \vdash e : \tau$ and $S;\Gamma \vdash \tau :: \kappa$ are the typing and kinding judgments for FINE (with refinement types) as defined in our prior work (Swamy et al. 2010). Informally, our theorem states that for well-formed environments, any source term e well-typed at τ, is translated (in a translated environment) to a term e' well-typed at type τ', where τ' is the translation of τ. A similar result holds for the derefinement of types. The proof of the theorem proceeds by mutual induction over the structure of the type- and term-derefinement judgments.

Theorem 1 (Derefinement preserves types)**.** *For any well-formed context $S;\Gamma;X$, there exists a context $S';\Gamma';X$ that is the derefinement of $S;\Gamma;X$; such that, for any source expression e well-typed at type τ, there exists an expression e' and type τ' that is the derefinement of e and τ, where e' can be given the type τ'. Similarly, for any type τ_1 well-kinded at kind κ, there exists a type τ_1' that is the derefinement of τ_1, where τ_1' can be given the kind κ. That is,*

$$\forall S,\Gamma,X,e,\tau.\ wf(S;\Gamma;X)\ \wedge\ S;\Gamma;X\vdash e:\tau\ \Rightarrow$$
$$\exists S',\Gamma',e',\tau',b.\quad S;\Gamma\hookrightarrow S';\Gamma'\ \wedge\ S';\Gamma';X\vdash e\overset{b}{\hookrightarrow}e':\tau'\ \wedge$$
$$S';\Gamma'\vdash\tau\overset{b}{\hookrightarrow}\tau'::\kappa'\ \wedge\ S';\Gamma';X\vdash e':\tau'$$

$$\forall S,\Gamma,X,\tau_1,\kappa.\ wf(S;\Gamma;X)\ \wedge\ S;\Gamma;X\vdash\tau_1::\kappa\ \Rightarrow$$
$$\exists S',\Gamma',\tau_1',b.\quad S;\Gamma\hookrightarrow S';\Gamma'\ \wedge\ S';\Gamma'\vdash\tau\overset{b}{\hookrightarrow}\tau'::\kappa\ \wedge$$
$$S';\Gamma'\vdash\tau'::\kappa$$

4. Translating FINE to DCIL

This section presents DCIL, an extension of a functional fragment of CIL. We use CIL generics to translate many basic FINE constructs (Kennedy and Syme 2004). DCIL extends CIL with affine types, type-level functions, and classes parameterized by values. We discuss how to represent all our extensions in standards-compliant .NET assemblies. Code consumers can choose to use a type checker for DCIL for security checking, but otherwise can run DCIL programs on stock .NET virtual machines.

4.1 Syntax

Figure 6 shows the syntax of DCIL. We re-use metavariables from FINE for syntactic categories in DCIL—the context will make the distinction clear. We write $\overline{\rho}$ for a finite-length sequence of ρ items, and $\overline{\rho}_n$ for an n-length sequence. Modules in FINE are translated to a combination of assemblies, modules, and inner classes in DCIL, where we use visibility qualifiers to model information-hiding in DCIL—this is discussed in our technical report.

module	mod.	::=	$\{\overline{\text{td}}, \overline{\text{dd}} \text{ in } e\}$
abs. class	td	::=	$T\langle\overline{\alpha::\kappa}, \overline{x:\tau}\rangle::\kappa\{\overline{\text{fd}}, \overline{\text{md}}\}$
data class	dd	::=	$D\langle\overline{\alpha::\kappa}, \overline{x:\tau}\rangle{:}T\langle\overline{\tau}, \overline{v}\rangle\{\overline{\text{vc}}, \overline{\text{fd}}, \overline{\text{md}}\}$
constraints	vc	::=	$x \doteq v$
fld. decl.	fd	::=	$f{:}\tau$
meth. decl.	md	::=	$\tau\ m\langle\alpha::\kappa\rangle(x{:}\tau)\{e\}$
value	v	::=	$x \mid D\langle\overline{\tau}, \overline{v}\rangle$
expr.	e	::=	$v \mid v.f \mid v.m\langle\tau\rangle(v)$
		$\mid$	x isinst $D\langle\overline{\tau}, \overline{v}\rangle$ then e_t else e_f
		$\mid$	let $x = e_1$ in e_2
type	τ	::=	$\alpha \mid T\langle\overline{\tau}, \overline{v}\rangle \mid\ !\tau \mid \backslash x{:}\tau_1.\tau_2 \mid \tau\ v$
kind	κ	::=	$\star \mid \mathtt{A} \mid \tau \to \kappa$

Figure 6. Syntax of DCIL

DCIL distinguishes two types of classes. All (non-primitive) types in FINE are translated to abstract classes T. FINE values $v{:}\tau$ are translated to instances of *data classes* D, where D extends T, the class corresponding to τ. Classes can be parameterized by a list of type parameters $\overline{\alpha::\kappa}$ and also by a list of value parameters $\overline{x:\tau}$. Both kinds of classes include field and method declarations, although bodies of method declarations in T-classes are empty. Data classes include value constraints $\overline{\text{vc}}$, which are analogous to FINE's pattern matching assumptions—we discuss these shortly.

Like FINE, the syntax of expressions in DCIL is presented in A-normal form. Expressions include values v (variables or instances of data classes D), field projections, method calls, and a runtime type-test construct, (v isinst $D\langle\overline{\tau}, \overline{v}\rangle$ then e_t else e_f). Let-bindings are syntactic sugar for initialization of (immutable) local variables in CIL. Both let-bindings and type-tests are macro instructions in DCIL—each corresponds to several CIL instructions. Types include type variables and fully instantiated abstract classes $T\langle\overline{\tau}, \overline{v}\rangle$. Affinely qualified types are written $!\tau$, as in FINE. DCIL includes a restricted form of type-level functions (written $\backslash x{:}\tau_1.\tau_2$) to represent dependent types. Type-level function application is denoted $\tau\ v$. Kinds include $\star$ and $\mathtt{A}$ as in FINE, and $\tau \to \kappa$, the kind of type-level functions.

4.2 Overview of DCIL

DCIL contains three main innovations. First, in addition to $\star$-kinded type parameters, classes can include affine types, type-functions, and values as parameters. Importantly, DCIL does not include type parameters of kind $\star \to \kappa$ or $\mathtt{A} \to \kappa$, a fundamental restriction of .NET generics which we aim to preserve. A violation of this property likely requires sweeping changes to CIL, contrary to our aim of accommodating affine and dependent typing using only the existing metadata facilities provided by .NET. In our approach, value parameters are represented using standard field declarations and type functions are encoded using custom attributes, but, ignoring these attributes still yields a valid .NET assembly.

Our second main contribution is a formalization of affine typing for DCIL. The mixture of affine and dependent typing is subtle and can require tracking affine assumptions in types as well as terms. Our formulation is streamlined by a crucial design element of DCIL—the separation of classes that represent source-level types (abstract classes $T\langle\overline{\tau}, \overline{v}\rangle$) from data classes ($D\langle\overline{\tau}, \overline{v}\rangle$). This separation makes sure that affine values never appear in types, much as in the source language, greatly simplifying the metatheory of DCIL. Affine types can be represented in CIL using .NET type modifiers—these are opaque to the .NET runtime, and only need to be interpreted by a DCIL-aware bytecode verifier.

Finally, we retain separate compilation of DCIL classes by augmenting the declaration of data classes with value constraints. For an intuitive sense of why separate compilation of DCIL classes poses a difficulty, consider the following source program fragment:

match b **with** true → λy:int. ((Refl_bool b) :proof (Eq_bool b true))

When typing this program in FINE, we can convert the type of Refl_bool b from proof(Eq_bool b b) to proof(Eq_bool b true), since the sub-term Refl_bool b appears in a context where b=true. However, when translated to DCIL, the lambda-expression is closure converted, and then translated to some data class D with a single value parameter b. To type check D, we need to ensure that it is only constructed in a context where its value parameter b can be proved equal to true. Value constraints in DCIL serve just this purpose—they record constraints about a class's value parameters so that the class can be checked independently of other classes; at every construction site of a class, we check that its value constraints are satisfied.

4.3 Static semantics of DCIL

Figure 7 shows several rules from the key judgments in the static semantics of DCIL. Derivations use a context Σ that collect declarations of both D- and T-classes; Γ, a local typing environment; and X a context containing usable affine assumptions.

The (WF-dd) rule defines well-formedness of a data class declaration. We include it here primarily to point out the scoping rules for the type and value parameters of a class declaration. In the first premise, we check that the kind κ_i assigned to each type parameter α_i is well-formed. We permit functional dependences among the kinds: in the first premise of (WF-dd) we check each κ_i in a context extended with the prefix of previous type parameters $\alpha_1::\kappa_1, \ldots, \alpha_{i-1}::\kappa_{i-1}$. For example, $\mathsf{DepArrow}\langle\alpha_1::\star, \alpha_2::\alpha_1 \to \star\rangle$ is a valid class declaration in DCIL, where the first type parameter α_1 appears in the kind of the second parameter. (DepArrow is used to represent dependent functions from FINE—cf. §4.4.) Similarly, we allow dependences among the value parameters $x_i{:}\tau_i$ (the second premise of (WF-dd)). The remaining premises check that the super-class $T\langle\overline{\tau'}, \overline{v'}\rangle$, each of the value constraints, field declarations, and method declarations are all well-formed. An important aspect of the last premise of (WF-dd) is that each method declaration is checked with its own set of affine assumptions X_i (disjoint from others) drawn from the value parameters of the class.

Note also that, in the last premise of (WF-dd), the method declarations are checked with the class's value constraints $\overline{\text{vc}}$ in the context—the key to enable separate type checking of DCIL classes. For this rule to be sound, we need to check that the constraints hold true at every construction site of the class. This check is handled by (T-New). In the first premise of (T-New), we lookup the constructed class's declaration in the signature Σ. In the second premise, we check that the value constraints $\overline{\text{vc}}$ are valid for the actual arguments $\overline{v}$ used to construct the class. The third premise of (T-New) checks the type arguments $\overline{\tau}$ against their expected kinds $\overline{\kappa}$. Since the scoping rules allow dependences among the type arguments, when checking the ith argument, we substitute the prefix of arguments for the bound type variables in the expected kind κ_i—we write $[\overline{\tau}/\overline{\alpha}_{i-1}]$ for the substitution $[\tau_1/\alpha_1 \ldots \tau_{i-1}/\alpha_{i-1}]$. The last premise of (T-New) is similar, but must account for dependences among the value parameters.

The (y isinst $D\langle\overline{\tau}, \overline{x}\rangle$ then e_t else e_f) form is DCIL's equivalent of FINE's **match** construct, where the data class $D\langle\overline{\tau}, \overline{x}\rangle$ plays the role of a pattern. This instruction is a macro that expands to multiple CIL instructions, where in the **then**-branch we include projections of each of the fields corresponding to the value parameters of y, the expression being scrutinized. DCIL provides no other way to project the value parameters of a data class. When checking this expression (T-Inst), we split the affine assumptions X, X' between the value y being scrutinized and the branches. We check the pattern and the true-branch in a context Γ' that includes bindings for

$\Sigma; \Gamma; X \vdash e : \tau$, $\Sigma \vdash \circ$ **wf** ... where $\Sigma ::= D{:}\mathrm{dd}, \Sigma \mid T{:}\mathrm{td}, \Sigma \mid \cdot$, $\Gamma ::= x{:}\tau, \Gamma \mid \alpha{::}\kappa, \Gamma \mid x \doteq v, \Gamma \mid \cdot$ and $X ::= \cdot \mid x, X \mid \cdot$

$$\frac{\forall i.\Sigma, \overline{\alpha{::}\kappa}_{i-1} \vdash \kappa_i \ \textbf{wf} \quad \Sigma; \overline{\alpha{::}\kappa}, \overline{x{:}\tau}_{i-1} \vdash \tau_i :: \kappa'_i \quad \Gamma = \overline{\alpha{::}\kappa}, \overline{x{:}\tau} \quad \Sigma; \Gamma \vdash T\langle\overline{\tau'}, \overline{v'}\rangle \ \textbf{wf} \quad \forall i.\Sigma; \Gamma \vdash \mathrm{vc}_i \ \textbf{wf} \quad \forall i.\Sigma; \Gamma, \overline{\mathrm{vc}} \vdash \mathrm{fd}_i \ \textbf{wf} \quad \overline{x} = \overline{X} \quad \forall i.\Sigma; \Gamma, \overline{\mathrm{vc}}; X_i \vdash \mathrm{md}_i \ \textbf{wf}}{\Sigma \vdash D\langle\overline{\alpha{::}\kappa}, \overline{x{:}\tau}\rangle : T\langle\overline{\tau'}, \overline{v'}\rangle\{\overline{\mathrm{vc}}, \overline{\mathrm{fd}}, \overline{\mathrm{md}}\} \ \textbf{wf}}\ (\text{WF-dd})$$

$$\frac{\Sigma; \Gamma; X \vdash y : \tau_1 \quad \Gamma' = \Gamma, \overline{x{:}\tau} \quad \Sigma; \Gamma'; \overline{x} \vdash D\langle\overline{\tau}, \overline{x}\rangle : \tau'_1 \quad \mathrm{unify}(\tau_1, \tau'_1) = \overline{\mathrm{vc}} \quad \Sigma; \Gamma', \overline{\mathrm{vc}}, y \doteq D\langle\overline{\tau}, \overline{x}\rangle; X' \vdash e_t : \tau \quad \Sigma; \Gamma; X' \vdash e_f : \tau}{\Sigma; \Gamma; X, X' \vdash y \ \mathrm{isinst}\ D\langle\overline{\tau}, \overline{x}\rangle \ \mathrm{then}\ e_t \ \mathrm{else}\ e_f : \tau}\ (\text{T-Inst})$$

$$\frac{\Sigma(D) = D\langle\overline{\alpha{::}\kappa}, \overline{x{:}\tau'}\rangle{:}\tau\{\overline{\mathrm{vc}}, \ldots\} \quad \Sigma; \Gamma \vdash \overline{\mathrm{vc}}[\overline{v}/\overline{x}] \quad \forall i.\Sigma; \Gamma \vdash \tau_i :: \kappa_i[\overline{\tau}/\overline{\alpha}_{i-1}] \quad \forall j.\Sigma; \Gamma, X_j \vdash v_j : \tau'_j[\overline{\tau}/\overline{\alpha}][\overline{v}/\overline{x}_{j-1}]}{\Sigma; \Gamma; X_1 \ldots X_m \vdash D\langle\overline{\tau}, \overline{v}\rangle : \tau[\overline{\tau}/\overline{\alpha}][\overline{v}/\overline{x}]}\ (\text{T-New})$$

$$\frac{\Sigma; \Gamma; X \vdash v : T\langle\overline{\tau'}, \overline{v'}\rangle \quad \Sigma(T\langle\overline{\tau'}, \overline{v'}\rangle) = \tau_2\ m\langle\alpha{::}\kappa\rangle(x{:}\tau_1) \quad \Sigma; \Gamma \vdash \tau :: \kappa \quad \Sigma; \Gamma; X' \vdash v' : \tau_1[\tau/\alpha]}{\Gamma; X, X' \vdash v.m\langle\tau\rangle(v') : \tau_2[\tau/\alpha][v'/x]}\ (\text{T-App})$$

$\Sigma; \Gamma \vdash \tau :: \kappa$ — Kinding of types

$$\frac{\Sigma; \Gamma \vdash \tau_1 :: \star \quad \Sigma; \Gamma, x{:}\tau_1 \vdash \tau_2 :: \kappa}{\Sigma; \Gamma \vdash \backslash x{:}\tau_1.\tau_2 :: \tau_1 \to \kappa}\ (\text{TK-Fun}) \qquad \frac{\Sigma; \Gamma \vdash \tau :: \tau_1 \to \kappa \quad \Sigma; \Gamma; \cdot \vdash v : \tau_1}{\Sigma; \Gamma \vdash \tau\ v :: \kappa}\ (\text{TK-App})$$

$$\frac{\Sigma(T) = T\langle\overline{\alpha{::}\kappa}, \overline{x{::}\tau'}\rangle{::}\kappa' \quad \forall i.\Sigma; \Gamma \vdash \tau_i :: \kappa_i[\overline{\tau}/\overline{\alpha}_{i-1}] \quad \forall j.\Sigma; \Gamma; \cdot \vdash v_j : \tau'_j[\overline{\tau}/\overline{\alpha}][\overline{v}/\overline{x}_{j-1}]}{\Sigma; \Gamma \vdash T\langle\overline{\tau}, \overline{v}\rangle :: \kappa'}\ (\text{TK-T})$$

$\Sigma; \Gamma \vdash \tau \cong \tau'$ and $\Sigma; \Gamma \vdash v \cong v'$ — Equivalence of types and values

$$\frac{}{\Sigma; \Gamma \vdash (\backslash x{:}\tau.\tau')\ v \cong \tau'[v/x]}\ (\text{TE-Beta}) \qquad \frac{\forall i.\Sigma; \Gamma \vdash \tau_{1,i} \cong \tau_{2,i} \quad \forall j.\Sigma; \Gamma \vdash v_{1,j} \cong v_{2,j}}{\Sigma; \Gamma \vdash T\langle\overline{\tau_1}, \overline{v_1}\rangle \cong T\langle\overline{\tau_2}, \overline{v_2}\rangle}\ (\text{TE-Refine}) \qquad \frac{v_1 \doteq v_2 \in \Gamma}{\Sigma; \Gamma \vdash v_1 \cong v_2}\ (\text{VE-Refine})$$

Figure 7. Static semantics of DCIL (selected rules)

the pattern-bound variables. As a result of y matching the pattern, we can deduce a number of equalities. These equalities, $\overline{\mathrm{vc}}$, are computed (in the fourth premise) by unifying the type τ_1 of y with the type of the pattern τ'_1. For example, if τ_1 is $\mathsf{Cred}\langle\mathsf{Alice}\rangle$ and τ'_1 is $\mathsf{Cred}\langle x\rangle$, the value constraints $\overline{\mathrm{vc}}$ include $x \doteq \mathsf{Alice}$. The true-branch, e_t, can use these equalities, in addition to an assumption that y matches the pattern, i.e., $y \doteq D\langle\overline{\tau}, \overline{x}\rangle$.

Finally, we show (T-App), the rule for method calls, which captures both type and term application in FINE. In the first premise, we type the receiver object v. Note that v's type is always an abstract class T, even though v is a constructed using a data class D. The second premise looks up the method declaration; the third checks the type argument; and the last premise checks the value argument. In the conclusion, as is standard with dependent typing, we substitute the actual v' for the formal x in the return type.

In the kinding judgment, (TK-Fun) defines the well-formedness of type-level functions. The first premise ensures that type-level functions can only receive non-affine values as arguments. This restriction, together with the separation of data classes D from type classes T, ensures that we do not have to track usages of affine assumptions at the type level. Application of type-level functions is handled by (TK-App), where the second premise shows the value v typed without any affine assumptions X. Finally, (TK-T) shows the kinding rule for abstract classes. As with (T-New), the dependences among the type and value parameters require that we apply prefix substitutions to the expected kinds and types. Also, as in (TK-App), the last premise of (TK-T) makes no use of affine assumptions.

The bottom part of Figure 7 shows selected rules from DCIL's type equivalence judgment. The complete relation is the reflexive, symmetric, transitive closure of the rules shown. The typing judgment is free to appeal to this relation to convert types of expressions at any point in a derivation—this makes our presentation of the typing judgment for DCIL non-syntax-directed. To be syntax directed, our implementation relies on annotations inserted by the FINE type checker to determine where in a derivation type equivalence is needed. The rule (TE-Beta) equates types related by β-reduction of type-level function applications. Type-level functions are essentially drawn from the simply-typed lambda calculus and, as such, are strongly normalizing. Thus, despite allowing computation in types, DCIL type checking remains decidable. (TE-Refine) lifts the equivalence relation into the type and value parameters of a class. Finally, (VE-Refine) equates value parameters v_1 and v_2 when $v_1 \doteq v_2$ is in the context.

Theorem 2 below establishes that DCIL is sound. The dynamic semantics of DCIL is formulated (like FINE) to account for affine typing. The small-step reduction relation for DCIL is written $\Sigma \vdash (M, e) \rightsquigarrow (M', e')$. Values with affine types are held in a mutable store M, where reads and writes to the store are destructive. Following a methodology adopted in our prior work (Swamy et al. 2010), Theorem 2, in addition to showing that well-typed programs never get stuck, guarantees that DCIL programs never destruct affine values more than once.

Theorem 2 (Soundness of DCIL)**.** *For all well-formed signatures Σ; environments Γ; non-values e; and stores M typeable with $\Sigma; \Gamma$, the following statements are true:*

1) *If $\Sigma; \Gamma; dom(M) \vdash e : \tau$ then there exists M', e' such that $\Sigma \vdash M, e \rightsquigarrow M', e'$.*

2) *If $\Sigma; \Gamma; X \vdash e : \tau$ and $\Sigma \vdash M, e \rightsquigarrow M', e'$ for some M', e', and $X \subseteq dom(M)$; then, there exists Γ', X' such that $\Sigma; \Gamma'; X' \vdash e' : \tau$ and M' is typeable with $\Sigma; \Gamma'$. 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 $X' = X \setminus \Delta_X$.*

4.4 Translation of FINE to DCIL

This section illustrates our translation from FINE to DCIL using several examples. The main subtleties arise in two parts of the translation. First, dependent functions are translated to instances of an abstract class DepArrow, overriding a single method App containing the translation of the function body. This idea is based on a scheme proposed by Kennedy and Syme (2004), who translate a polymorphic (non-dependent) lambda calculus to an object-oriented language like CIL. The primary novelty of our translation lies in the extension of this translation to capture the functional dependences introduced by dependent types in FINE. We further extend this mechanism to account for affine types. The second novelty of our translation relates to the computation of value constraints in data class declarations. These constraints are computed with the assistance of the source-level type checker and, as mentioned previously, enable separate compilation of DCIL classes.

Translation of type constructors. Type constructors are translated to declarations of abstract classes T. The type and value parameters of a type constructor are carried over directly. For example, the type of proofs, proof::$\star \rightarrow \star$, is represented in DCIL as an abstract class with a single type parameter: proof<α ::$\star$>::$\star$. Dependent type constructors like Eq_att:: att $\rightarrow$ att $\rightarrow \star$ are translated to abstract classes with value parameters: Eq_att<x:att, y:att>::$\star$.

Translation of data constructors. Data constructors in FINE are translated to declarations of data classes D that extend the abstract class corresponding to the type constructed by D. For example, the And_elim_1: proof<And $\alpha\,\beta$> $\rightarrow$ proof<α> data constructor is translated to the class,

And_elim_1<α ::$\star$, β ::$\star$, x:proof<And<α,β>>> : proof<α>

The value parameter of And_elim_1 corresponds to a field that holds a proof<And<α,β>> value, but notice that this value parameter does not appear in the type proof<α> constructed by And_elim_1. This is in contrast to the data constructors of dependent types. For example, the reflexivity axiom Refl_eq_att: a:att $\rightarrow$ proof (Eq_att a a) is translated to a data class Refl_eq_att<a:att>:proof<Eq_att<a,a>>. The value parameter of Refl_eq_att corresponds both to a single field declaration in the body of the class and additionally appears as an index in the type proof<Eq_att<a,a>> that it constructs.

Translation of function types. Dependent function types in FINE are translated to instances of the abstract class shown below:

DepArrow$\langle \alpha_1$::$\star, \alpha_2$::$\alpha_1 \rightarrow \star\rangle$ :: $\star\{\quad (\alpha_2\ x)$ App$(x$:$\alpha_1)\{\}\quad\}$

Class DepArrow takes two type parameters: α_1 for the argument type and α_2 for a *type function*—the return type of App is the result of applying α_2 to the argument x. Source-level types such as p:prin $\rightarrow$ cred p are translated to instances of DepArrow; in this case, PCredP:DepArrow$\langle$prin, \x:prin.cred$\langle x\rangle\rangle$. The App method of PCredP is of the form $((\backslash x$:prin.cred$\langle x\rangle)\ p)$ App$(p$:prin) (by instantiating types in the declaration of DepArrow). By the rule (TE-Beta) in the type equivalence relation, the return type of this method is cred<p>, analogous to the type returned by the source-level function. Each function type in FINE is translated to a distinct class (like PCredP) in DCIL and overrides the App method suitably. A closure conversion step collects the free variables of a function and adds these as additional type and value parameters to the class.

We also include the abstract classes shown below to represent non-dependent functions, and functions that take affine arguments or produce affine results. Notice that the second type parameter of Arrow_AA is not a type-function, since the type system ensures that affine values can never appear within types.

Arrow$\langle \alpha_1$::$\star, \alpha_2$::$\star\rangle$ :: $\star\{\quad (\alpha_2)$ App$(x$:$\alpha_1)\{\}\quad\}$
Arrow_AA$\langle \alpha_1$::A, α_2::A$\rangle$ :: $\star\{\quad (\alpha_2)$ App$(x$:$\alpha_1)\{\}\quad\}$
DepArrow_A$\langle \alpha_1$::$\star, \alpha_2$::$\alpha_1 \rightarrow$ A$\rangle$ :: $\star\{\quad (\alpha_2\ x)$ App$(x$:$\alpha_1)\{\}\quad\}$

Translation of dependent pairs. Dependent pairs are translated similarly to dependent functions. The abstract class DepPair_A below corresponds to the type of a dependent pair where the second component is affine—as with functions, we include variants of DepPair_A for pairs of other kinds.

DepPair_A$\langle \alpha_1$::$\star, \alpha_2$::$\alpha_1 \rightarrow$ A$\rangle$ :: A$\{\}$

Classes that represent pairs are just data containers with no methods at all. In contrast to functions, distinct dependent pair types in FINE can be translated to the same class in DCIL. We include data classes of the form shown below:

DA$\langle \alpha_1$::$\star, \alpha_2$::$\alpha_1 \rightarrow$ A, x:$\alpha_1, y : \alpha_2\ x\rangle$: DepPair$\langle \alpha_1, \alpha_2\rangle$

The data class DA can be instantiated appropriately to represent specific source values. For example, the source value (s, tok) of type (s:st $*$ Statels s) is translated to a new DA value using the constructor application DA$\langle$st, \s:st.Statels$\langle$s$\rangle$, s, tok$\rangle$ and is given the type DepPair_A$\langle$st, \s:st.Statels$\langle$s$\rangle\rangle$. As mentioned before, DCIL provides no way to project value parameters—the only way to destruct a DepPair_A class is by using the isinst construct.

Using value constraints. To illustrate the use of value constraints, consider, once again, the source term:

match b **with** true $\rightarrow$ λy:int. ((Refl_bool b) :proof(Eq_bool b true))

The lambda expression is translated to a class D<b:bool>, a subclass of Arrow<int, proof<Eq_bool<b, true>>>, where closure conversion adds the free variable b as a value parameter of D. The body of the lambda-term is translated to the body of the overriden App method of D, where, to give the sub-term Refl_bool<b> the type proof<Eq_bool<b,true>>, the DCIL checker needs an assumption b $\doteq$ true. The translation from FINE to DCIL records this assumption (provided by the **match** on b) as a value constraint in the declaration of D. The **match** statement is itself translated to an isinst statement in DCIL, and in its then-branch, where the D-class is constructed, we check that the constraint b $\doteq$ true holds.

4.5 Type-preserving translation

Theorem 3 below states that the translation preserves types—$\|\cdot\|$ denotes the translation of environments, and the judgments that use $\twoheadrightarrow$ stand for translation of types and terms.

Theorem 3 (Type-preserving translation)**.** *Suppose, for a well-formed source environment $S; \Gamma; X$, and for a source expression e_{src} well-typed at type τ_{src} (i.e., $S; \Gamma; X \vdash e_{src} : \tau_{src}$) we have that e_{src} is derefined (for some b) to e'_{src} at type τ'_{src} (i.e., $S; \Gamma; X \vdash e_{src} \overset{b}{\hookrightarrow} e'_{src} : \tau'_{src}$). Then, there exists a target expression e_{tgt} and class declarations Σ, such that e'_{src} is translated to e_{tgt} and Σ (i.e., $\|S\|; \|\Gamma\| \vdash e'_{src} \twoheadrightarrow e_{tgt}; \Sigma$); and there exists a target type τ_{tgt} such that type τ'_{src} is translated to τ_{tgt} (i.e., $\|S\|, \Sigma; \|\Gamma\| \vdash \tau'_{src} \twoheadrightarrow \tau_{tgt}$). Furthermore, e_{tgt} is well-typed at type τ_{tgt}, i.e., $\|S\|, \Sigma; \|\Gamma\| \vdash e_{tgt} : \tau_{tgt}$.*

5. Implementation

This section describes the implementation of our prototype compiler and our experience using it on several small programs. Our compiler is implemented in approximately 20,000 lines of F# code, extending the parser and the binary writing libraries of the F# compiler. Our application programs are, for the most part, reference monitors—security-critical kernels of applications that are expected to be compact. These programs enforce many kinds of policies, including those based on security automata, information flow controls, and role- and history-based authorization. Our measurements show that, currently, the cost of carrying proofs can increase the size of binaries by more than an order of magnitude. However, despite their large size, type checking DCIL assemblies is fast. We also report on an experiment with our largest benchmark, where, through the use of a custom solver, we were able to reduce code size by more than a factor of 25. In summary, our results indicate that end-to-end verification is possible for programs that use common security policies, and, with improvements in certifying solvers, the overhead of carrying proofs can be made practical.

5.1 Application programs

Figure 8 shows the results of our compiler on six example programs. The columns from left to right are the name of the program; the number of lines of source code (LOC); the time (in seconds) for parsing and type checking source programs without extracting proofs (SC); the time to extract proofs and to derefine (DR); the time for translating to DCIL (Trans); the time to type check target programs (TC); the size in bytes of .NET assemblies that do not

Name	LOC	SC	DR	Trans	TC	NoPf	Pf
AuthAC	34	0.36	0.56	0.25	0.08	20K	30K
Automaton	121	1.53	0.76	0.41	0.09	20K	40K
iFlow	127	2.90	16.9	17.8	0.42	30K	840K
HealthWeb	318	2.82	47.4	65.7	1.14	80K	2.1M
Lookout	519	4.99	54.3	37.8	0.71	120K	1.8M
ConfRM	647	10.1	68.5	81.2	1.47	110K	3.3M
Total	1766	22.7	188.42	203.26	3.91	380K	8.3M
ProofLib	9943	20.4	55.84	577.8	6.73	51.3M	51.3M
Conf(Z3)	177	6.83	55.4	62.9	1.25	45K	2.4M
Conf(SS)	177	6.83	37.8	0.39	0.11	45K	83K

Figure 8. Compilation times and binary sizes on benchmarks

include proof terms (NoPf); and the size of assemblies that do include proofs (Pf). Our experiments were performed on a 3.2 GHz Pentium Core Duo running Windows Vista.

Standalone benchmarks. Our simplest benchmark is AuthAC, which implements a password-based authentication mechanism combined with a group-based access control policy. Proving the correctness of AuthAC requires constructing a single proof term showing that a principal making a request for a resource is a member of the appropriate group. Automaton is more interesting—it enforces a protocol on file system resources specified as a security automaton. It uses refinement formulas to reason about the equivalence of file handle aliases, and combines this with affine and dependent typing to model the current state of a file. iFlow implements a canonical lattice-based information flow policy, with dynamic security labels (Zheng and Myers 2004). Types in iFlow are refined using a proposition CanFlow l m, where l and m are security labels. We use runtime tests of dynamic labels together with user-defined assumptions that defines the label lattice in order to discharge proofs of the CanFlow proposition.

HealthWeb is a reference monitor for an application that manages a database of electronic medical records. It enforces a stateful authorization policy, where the authorization state records attributes like role activations, current relationships between doctors and patients, and patient consent directives. Patient records are classified by subject which, together with the authorization state, controls the privilege to read, write, delete, annotate, or search for records. The reference monitor serves requests from a web-based front-end written in ASP.NET and C#, and provides a secure interface to a SQL Server database with an object-relational mapping implemented using F#. As a server-side program, we anticipate that the proof terms produced for the verification of HealthWeb could be logged at runtime to construct audit trails of authorization decisions; however, we have yet to implement such an auditing facility.

ConfRM is a reference monitor based on Continue, a widely used conference management tool (Krishnamurthi 2003). This application was previously implemented and described in detail in our prior work Swamy et al. (2010) and is currently our largest benchmark. It enforces a stateful authorization policy that is divided into 9 temporal phases and manages 12 different kinds of privileges.

Lookout is a larger version of the example described in §2. Two additional features of Lookout are of particular interest. First, in addition to the stateful authorization policy shown in §2, we provide facilities to track information flows through plugin code. For example, rather than return a string, the readEmail function from Figure 1 returns a value of the abstract type labeled string (Email e)—the label Email e records the provenance of the string, namely, that it originated from the email e. User policies can refer to these labels to specify information flow controls. For example, one of our example user policies prevents plugins from replying to an email with content derived from other, more sensitive, emails.

Second, we provide a way for plugins to selectively share information with each other via the cookie store. Rather simply asserting (using Permit p a) that a principal p holds the privilege to perform an action a, our implementation uses a policy in which a privilege is granted by one principal to another, e.g, Permits p q a records a permission granted by p to q to perform the action a. We use this decentralized model of permissions to build secure plugin mashups. When placing a cookie in the store, a plugin can register an function closure that mediates access to that cookie. When the reference monitor returns a cookie to the principal p, we check that the returned value has the type {c:cookie | Derivable (Permits owner p (ReadCookie c.name)) s}), indicating that the cookie's owner authorizes p to read the cookie.

ProofLib is an auto-generated library of commonly used (verified) lemmas that assist with translation of Z3 proof terms. Z3 proofs often use rewriting steps that may, for example, rearrange the order of clauses in a formula. Or, a proof may use a number of variants of a rule to eliminate double negation. Rather than reconstruct proofs of these steps each time, proof terms simply use lemmas exported by the ProofLib module.

5.2 Compilation times and producing smaller proofs

In general, our measurements show that type checking DCIL programs is fast. For example, typechecking the 51MB ProofLib takes less than 7 seconds. However, the last two columns of Figure 8 show that the increase in code size due to proof terms can be quite substantial—21x on average, as much as 53x in some cases (Conf(Z3)). Clearly, this is much larger than we would like. Large proofs contribute to the bulk of the total compilation time for our application programs, both in derefinement which must synthesize these proofs, and in the translation from FINE to DCIL. When actively developing code, we often use a "source-checking only" mode for quicker feedback (the SC column). This mode typechecks source programs and uses Z3 to decide refinement formulas, but does not extract proofs.

The overhead due to proofs can be much lower with appropriate support from an external solver. The last two rows of Figure 8 are Conf(Z3) and Conf(SS). These are identical programs representing the main event loop of ConfRM, where much of the verification burden lies. The Conf(Z3) row shows measurements for this program compiled with all proofs produced by Z3. The Conf(SS) line shows compilation results for this benchmark where all proofs were generated using a simple, unification-based first-order solver that we wrote for this purpose. Our measurements show that our simple solver can produce proofs that are 25 times smaller than Z3 proofs. However, our simple solver is not nearly as full-featured as Z3 and can only produce proofs by repeated application of and-introduction and elimination, quantifier instantiation, and modus ponens. Getting all of Conf's proof obligations to fall into this fragment required some careful rewriting—so, our simple solver is in no way a substitute for Z3.

A closer examination of Z3's proofs for Conf(Z3) suggests a few reasons why its proofs are so big. First, proofs contain a number of steps that pertain to manipulating the structure of quantified formulas. A first-order solver that used a more direct treatment of quantification is likely to produce more compact proofs. Second, SMT solvers have for long been optimized for speed rather than proof size. For example, a number of proof steps reported by Z3 involve rewriting formulas into specific normal forms since these are conducive to faster proof search. However, each of these rewrite steps has to be translated in to a proof term. Finally, proofs occasionally contain truly redundant steps, e.g., proofs of formulas that have already been assumed. Our proof extraction modules attempt to detect and discard such steps. However, there remain several opportunities to post-process Z3 proofs to produce smaller proof terms—we plan to investigate this in future work.

6. Related work

This section discusses related work not already covered elsewhere in this paper. Our approach of compiling FINE to DCIL is an instance of proof-carrying code (PCC) (Necula 1997) and typed assembly language (TAL) (Morrisett et al. 1999). Traditionally, both TAL and PCC have been applied to prove the memory safety of assembly language programs, rather than for security verification of bytecode. More recently, Yu and Islam (2006) have proposed a typed assembly language for confidentiality and prove that it enforces a noninterference property. Also related is Barthe et al. (2007) type system for noninterference for Java bytecode. Barthe et al. provide a formally certified implementation of their bytecode verifier by extracting an implementation from Coq. Their bytecode language also includes features like exceptions, which are omitted from DCIL. However, both these systems focus solely on checking the enforcement of information flow policies. In contrast, DCIL provides general support for dependent and affine types at the bytecode level, rather than building in special support for information flow policies. Our prior work Swamy et al. (2010) shows that both information flow policies and policies like stateful authorization can be enforced in FINE. Our type preservation result extends this result to DCIL. Additionally, both Barthe et al. and Yu and Islam's systems only enforce information flow policies with static security labels. Dependent types in DCIL allow us to enforce information flow policies with dynamic labels (Zheng and Myers 2004), and we put this to good use in our implementation of Lookout.

Dependently typed object-oriented programming languages have been studied previously. For example, the X10 programming language (Nystrom et al. 2008) and the HOOP calculus (Flanagan et al. 2006), use dependent types to state invariants on object-oriented programs. However, both of these are source languages, whereas DCIL is a bytecode language. X10 and HOOP also have imperative features; DCIL is functional, but uses affine types to model mutable state.

Refinement typing in FINE is closely related to similar constructs in F7 (Bengtson et al. 2008). Our work was designed, in part, to be directly applicable to F7, which like FINE, is also based on F#. In the future, we plan to investigate using our tools to certify the compilation of F7 programs that have been verified to correctly implement a number of cryptographic authentication protocols. Like F7, the Sage language (Flanagan 2006) also uses a *trusted external solver* to discharge proofs of refinement formulas, but automatically insert runtime checks when the prover fails to discharge a proof obligation. Failed runtime checks can cause subtle leaks of information, and so automatic insertion of runtime checks is not yet a feature of our compiler, where security is the primary concern.

Concurrent with our work, Böhme (2009) has implemented a tool to verify Z3 proofs in Isabelle/HOL. As discussed in §3.2, proof terms in FINE cannot make use of higher-order logic, due to constraints imposed by the type system of CIL. Relying only on first-order constructors for proofs complicates our proof extraction libraries, and also requires a larger proof kernel to represent specialized axioms about equality at each type.

7. Conclusions

This paper has presented a type-preserving compiler that translates FINE, a source-level programming language for enforcing rich security policies, to DCIL, a new extension of the bytecode language for the .NET virtual machine. We have used our compiler to construct and verify the security-critical modules of a number of applications. Although verification for DCIL is already relatively fast, we anticipate further improvements to come as proofs produced by solvers become more compact. As such, our work makes it possible for developers to use a high-level language to program security-critical code, and for end-users to receive formal proofs that the code they rely on is secure.

Acknowledgments. We thank Trishul Chilimbi, Jeremy Condit, and several anonymous reviewers for their helpful comments on an earlier version of the paper; Shriram Krishnamurthi for providing us with Continue's policy; Nikolaj Bjørner and Leonardo de Moura for help with Z3; and Karthik Bhargavan, Johannes Borgstroem, Cédric Fournet, and Andy Gordon for numerous discussions about this work.

References

A. W. Appel and E. W. Felten. Proof-carrying authentication. In *CCS*. ACM, 1999.

K. Avijit, A. Datta, and R. Harper. Distributed programming with distributed authorization. In *TLDI*. ACM, 2010.

G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight noninterference Java bytecode verifier. In *ESOP*. Springer, 2007.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*. IEEE, 2008.

Y. Bertot and P. Castéran. *Coq'Art: Interactive Theorem Proving and Program Development.* Springer Verlag, 2004.

S. Böhme. Proof reconstruction for Z3 in Isabelle/HOL. In *SMT Workshop*. Springer, 2009.

L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *TACAS*. Springer, 2008.

D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *LNCS*. Springer, 2006.

ECMA. Standard ECMA-335: Common language infrastructure, 2006.

C. Flanagan. Hybrid type checking. In *POPL*. ACM, 2006.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*. ACM, 1993.

C. Flanagan, S. N. Freund, and A. Tomb. Hybrid types, invariants, and refinements for imperative objects. In *FOOL/WOOD '06*, 2006.

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*. ACM, 2008.

A. Kennedy and D. Syme. Transposing F to C#: Expressivity of polymorphism in an object-oriented language. *Concurrency and Computation: Practice and Experience*, 16(7), 2004.

S. Krishnamurthi. The Continue server. In *PADL*. Springer, 2003.

R. Milner. LCF: A way of doing proofs with a machine. In *MFCS*, 1979.

G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM TOPLAS*, 21(3), 1999.

G. C. Necula. Proof-carrying code. In *POPL '97*. ACM, 1997.

N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *OOPSLA '08*. ACM, 2008.

A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 21(1):5–19, Jan. 2003.

A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified programming in Guru. In *PLPV*. ACM, 2008.

N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *S&P*. IEEE, 2008.

N. Swamy, J. Chen, and R. Chugh. End-to-end verification of security enforcement is fine. Technical Report MSR-TR-2009-98, MSR, 2009.

N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *ESOP*. Springer, 2010.

D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.

J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. In *CSF*. IEEE, 2008.

D. Yu and N. Islam. A typed assembly language for confidentiality. In *ESOP*. Springer, 2006.

L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *FAST '04*. Springer, 2004.