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

(Preliminary version)

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 still be compiled further to be executable on real computers. 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.

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 authenti-
contribution is a formalization of a source-to-source transformation that we dub derefinement—we explain its significance shortly.

A compiler implementation. We have completed an implementation of a compiler (publicly available) that translates FINE programs to executable .NET assemblies, verifiable using a typechecker for DCIL. A key component of our compiler is a module that extracts typeable proofs from Z3. 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 proofs for Z3.

Several application programs. We evaluate our compiler on several application programs written in FINE. Our main example targets security verification of third-party plugin modules using an approach related to proof-carrying code (Necula 1997). This application, called LOOKOUT, is a small plugin-based office utilities client. We present a reference monitor for this application written in FINE, as well as several plugins built for use with this application. 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. In addition to LOOKOUT, we also report on an application that manages a database of electronic health records, and a server for managing conference submissions.

Experimental evaluation. We report on experiments using our compiler on about 20,000 lines of code, of which nearly 2,000 lines are from application programs, 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 an assembly of around 50MB. 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 5x 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 these solvers produce.

1.2 Overview of our compilation chain

Source programmers use refinement and affine types in FINE to specify and enforce a range of security policies. For example, a simple access control policy on files may give the function `fread` the type `p:prin \rightarrow \text{cred}\ p \rightarrow \{\text{file} \mid \text{CanRead}\ p\ f\} \rightarrow \text{string}`. This is the type of a dependent function whose first argument is a principal, where the formal name `p` is bound to the right of the arrow. The next argument of type `\text{cred}\ p` is a credential authenticating `p`. The third argument is given the refined type `\{\text{file}\mid\text{CanRead}\ p\ f\}`, the type of all files `f` for which the proposition `\text{CanRead}\ p\ f` is true. FINE programs also include assumptions that define a security policy (e.g., `\text{assume}\ \text{CanRead}\ p\ \text{Alice}\ \{\text{"a.txt"}\}`), and the type checker uses an external solver to verify that the formulas that appear in refinement types are deductible from the assumptions.

The first phase of our compilation chain is a source-to-source transformation called derefinement, in which values with refined types are associated with proofs witnessing the deducibility of refinement formulas. For example, the type of `fread` is derefined to `p:prin \rightarrow \text{cred}\ p \rightarrow \text{file}\ \mid\text{CanRead}\ p\ f\) \rightarrow \text{string}`. Typeable proof terms are constructed automatically from Z3’s representation of proof objects. Proof terms are in the style of LCF (Milner 1979), i.e., these are built using the constructors of an abstract data type `pf`, where these constructors form a small trusted kernel that axiomatize the inference rules of a first-order logic.

The next phase involves a type-directed translation of derefined source programs to DCIL. The essence of DCIL is in the handling of dependent functions. Functions in FINE are translated to instances of an abstract class `DepArrow<\alpha:\star,\beta:\alpha\rightarrow\star>`. This is a class with two parameters, the first of which, `\alpha`, is an ordinary type parameter of kind `\star`, represented using standard .NET generics. In order to capture the functional dependences characteristic of dependent types, classes in DCIL can also be parameterized by `type-level functions`. The second parameter of `DepArrow` is a type-level function that constructs a type of kind `\star` from a value of type `\alpha`. For example, the derefined type of `fread` is translated (in part) to `DepArrow<\text{Prin}, p:Prin, DepArrow<\text{Cred}, p>, ...>`. Here, `\text{Prin}` is a class standing for the translation of the `\text{prin}` type in FINE; `\langle p, \text{Prin}, \text{DepArrow}<\text{Cred}, p>, ...\rangle` is a type-level function from `\text{Prin}` values to types; and `\text{Cred}\ p`, 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 `\text{cred}\ p`.

This additional type information can be represented using the standard metadata facilities of .NET. The type checker for DCIL is syntax directed and verifies programs without an external solver.

The extended version of this paper. Restrictions on the length of this paper prevent us from giving a complete treatment of a number of our results. Here, we summarize the contents of the supplementary material associated with this submission. The extended version of this paper includes complete formalizations of the static and dynamic semantics of DCIL, the translation from FINE to DCIL, the derefinement translation, and proofs of the theorems that appear 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’s type system is preserved in the translation to DCIL—we make no further mention of this result in this paper. Our extended report also includes a detailed description of some of the features of our LOOKOUT application omitted here, in particular, a treatment of information flow tracking in plugin code. Finally, the supplementary material also includes links to downloads for our compiler and example programs.

2. Programming in FINE

We begin by giving the reader a flavor of programming in FINE by presenting fragments from LOOKOUT, the plugin-based office utilities application described in the Introduction. Our type-preserving compiler provides assurances to an end-user that the reference monitor for LOOKOUT properly enforces her policy. A user can download DCIL assemblies for a plugin, type check it against a specific policy using a lightweight syntactic checker, and only install it if the check succeeds. LOOKOUT also 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.

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 refer-
1 module LookoutRM
2 type prin (* the type of principals *)
3 type cred :: prin → a (* a credential is a tuple (p, a) *)
4 private type email = {sender:prin, contents:string}
5 type envname = MsgIn | MsgOut | ComposeEmail | ...
6 type event :: envname → a → a
7 val mk_email :: prin → string → email
8 val sender::email → {p|p|msg.sender}
9 (* A vocabulary for an authorization policy *)
10 type action = Read :: 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 | Friend | Plugin | ...
16 type att = Role::prin → role → att
17 | HasRepliedTo::prin → email → att
18 type st = list att
19 (* An affine typed revokable signature of the program state *)
20 private type States :: st → A :: Sign :: st → States s
21 (* Propositions to define authorization constraints *)
22 type In :: att → st → a (* list membership *)
23 type Derivable :: perm → st → e (* dynamically derived perms *)
24 (* Useful type abbreviations *)
25 type ok <p, a, action> = {s|st | Derivable (Perm p a) s}
26 type plus <q, a> = {st | In a ∧ for all (b|att).In b s ⇒ In b x}
27 (* An API for plugins *)
28 val readEmail :: p|prin → cred p → e::email → sok<p, Read e> →
29 States s → (string → States s)
30 val replyTo :: p|prin → cred p → e::email → sok<p, ReplyTo e> →
31 States s → (s<ok,HasRepliedTo p e> × States s')
32 val setCookie :: p|prin → cred p → name:string → value:string →
33 sok<p, SetCookie p name → States s → States s
34 val getCookie :: p|prin → cred p → owner:prin → name:string →
35 sok<p, (ReadCookie owner name) > → States s →
36 (option string → States s)

Figure 1. A fragment of a reference monitor for LOOKOUT

rence 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. Finally, LOOKOUT provides functions to get and set values (“cookies”) from a key/value store. Our implementation augments the example 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. The technical aspects of FINe that enable such policies to be expressed have been established previously by Swamy et al. (2009). Here, we show that these constructs can be put to good effect in the construction of programs like LOOKOUT which are assembled from modules authored by multiple parties.

Figure 1 shows a fragment of the API exposed by the LOOKOUT reference monitor to plugins. Comments are shown italicized within (*...*) delimiters. The types given to this API specify authorization constraints; for example, looking ahead to the readEmail function on line 28, we see an argument sok<p, Read e> which represents a constraint that the caller p has the Read 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.

Lines 2-3 show the type of the principals prin and credentials cred p. The constructor cred is given the kind prin → a, indicating that it constructs a type of kind a, from a value of type prin; in other words, cred is a dependent type constructor. The concrete representation of principals and credentials is irrelevant to our example—we could, for example, use public keys or just user names and passwords.

Line 4 shows the type email. This type is marked as private—the FINe’s module system ensures that modules without the requisite privilege treat LookoutRM’s private types abstractly. 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 5-6 shows the type of event names envname and the type event e α, consisting of an event name n and some payload of type t generated when the event is triggered. Line 7 exposes a constructor for email, and, line 8 provides an accessor to examine the sender field of an email without restriction—access control will apply only to the contents field. 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:T | φ), where x is the formal name of a value of type T, and φ is bound in φ, a type that represents a first-order formula (with equality) (\(\phi\)).

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 name of the cookie’s key represented as a string.

The type att shown at line 18 forms the basis of the stateful authorization policy implemented by LOOKOUT. We adopt Swamy et al.’s model for stateful authorization, which in turn is based on 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, HasRepliedTo p e is used by the reference monitor 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: *, the kind of normal types, and &, the kind of affine types. Values with affine types may be used at most once. The notation Stats :: st → A indicates that Stats constructs an affine type from an st value. A value of State s is a signature from the reference monitor attesting that st s holds the current authorization attributes.

Next, at lines 22-23, we show two propositions with which to state authorization constraints in types. 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 st. Lines 25-26 show some convenient abbreviations that make use of these propositions to define refined types. The type ok <p, a> is a refinement of st to those values s 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 Read e privilege in the the current authorization state s. The return value of readEmail is a tuple 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, (st + t'), where s names value in the first component of the tuple and is bound in the type t'.
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. In contrast to Swamy et al. (2009) we adopt an A-normal presentation (Flanagan et al. 1993) of FINE. This helps to simplify the translation, as well as being convenient in 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 \( \tau \) include type variables, dependent functions, dependent pairs, quantified types, type constructors and their applications to types or values, types with affine qualifiers \(!\tau\) and \(\tau\), and refinement types. Types are classified according to the kind \( \kappa \), where \( *\) is that kind of unrestricted types, and \(\Lambda\) 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 \( \tau \) has kind \( *\)—indexing types with affine values is prohibited. Swamy et al. argue 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 programs. Programs are translated in the presence of a signature \( S \) that assigns kind and types to each type and data constructor; and a typing environment \( \Gamma \), which, in addition to variable bindings, records equality assumptions \( v \equiv v' \) 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 metavariable \( \phi \) for types that stand for formulas or proofs of formulas. The logical connectives in formulas are represented using the binding constructs, e.g., \( \And: * \rightarrow * \rightarrow * \), \( \Or: * \rightarrow * \rightarrow * \), \( \Not: * \rightarrow * \), and quantified formulas are represented using the binding constructs provided by dependent types. A universally quantified formula \( \forall (x: \tau). P x \) is represented as a dependent function \( x: \tau \rightarrow P x \), where \( P: \tau \rightarrow * \); existential quantification \( \exists (x: \tau). P x \) is represented using a dependent pair \( (x: \tau \rightarrow P)\).
We use an LCF-style (Milner 1979) proof system. An abstract datatye (ADT) \( pf :: \star \rightarrow \star \) represents 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 classical first-order logic with equality. User-provided assumptions constructors of this ADT represent inference rules that axiomatize a datatype (ADT)

The derefinement translation associates explicit proofs of formulas such as Operational Type Theory (Stump et al. 2008).

Figure 4 shows a few key rules from our derefinement judgment.

This judgment is written \( S; \Gamma \vdash \tau \leftrightarrow b \cdot \tau' :: \kappa \), and reads that in a context with a signature \( S \) and environment \( \Gamma \) (wherein all types have already been derefined), a source type \( \tau \) is derefined to \( \tau' \) of kind \( \kappa \). The superscript \( b \) is either bare or the constant box. In the latter case, this indicates that the type \( \tau \) was translated to a dependent pair of the form \( (x: \tau' \cdot pf \phi) \)—values of this type are “boxed” with a proof of the formulas \( \phi \).

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 so that they are in this form. In (D2), we show the translation of a function type, where the argument type \( \tau_1 \) is translated to the (unboxed) type \( \tau_1' \). 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. This serves two purposes—first, in the body of a function with this type, the argument \( x \) can be used at the type \( \tau' \); more importantly, the name \( x \) of type \( \tau' \) is bound in the return type \( \tau_2' \), where it may, for example, index another type. 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.

The derefinement of expressions has a similar form: \( S; \Gamma; X \vdash e \leftrightarrow b \cdot e' : \tau \), 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 \( m,e \), \( s,t \), 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 \( b \equiv \text{true} \). At the right we show the derefined program. Values that are given boxed types, like \( b' \), are unboxed immediately to bind both the underlying value and the proof in the context. The call to the function \( h::pf(\text{Derivable} \ p \ s) \rightarrow \text{Stmts} \ s \rightarrow \text{Stmts} \ s \) requires a proof term as its argument. The auto-generated proof term is shown as the value \( v \), shown enclosed in a box in the figure.

To discharge proofs, q our compiler constructs a first-order theory for Z3 by collecting user-provided axioms, variable bindings and match assumptions from the type environment \( \Gamma \), e.g., bindings for normal variable like \( b \), proof terms like \( pf1 \), and, in the then-

---

**Figure 4.** Selected rules from the derefinement of FINE types: \( S; \Gamma \vdash \tau \leftrightarrow b \cdot \tau' :: K \), where \( b ::= \text{bare} \mid \text{box} \)
branch of the conditional, the assumption \( b = \text{true} \). We then interpret
the negation of the goal, (e.g.,
This section presents
4. Translating

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

3.4 Generating proof terms

Consider typing the program of Figure 5 in the presence of the user-provided assumption:

\[
\text{assume } U \text{forall} (s,\text{st}). \ln (\text{Role me Plugin}) s' \Rightarrow \text{Derivable} \ p \ s'.
\]

Note that \( p,\text{perm} \) is bound in the context. At the call to readFoo, we are required to construct a proof term with type \( \text{pf(Derivable \ p \ s)} \). We show a proof term below (omitting type instantiations for clarity), where \( U \) is the data constructor shown below:

\[
\begin{align*}
\text{U,pf} & \equiv (s,\text{st} \to \text{pf} (\ln (\text{Role me Plugin}) s')) (\text{Derivable} \ p \ s')). \\
\text{Bind} & \equiv \lambda (s,\text{st} \to \text{pf} (\ln (\text{Role me Plugin}) s')) (\text{Derivable} \ p \ s')). \\
\text{MP} & \equiv (\text{MP} (\text{pf(Ref\_bool \ b);pf(Eq\_bool \ b \ true)}) (\text{iff ions} \ p)) (e). \\
\text{We use } & \text{iff ions} \ p \text{ to obtain a proof of the forward direction of the bidirectional implication. We then apply the modus
ponens constructor } \text{MP \ to a proof of } \text{pf(Eq\_bool \ b \ true).}
\end{align*}
\]

Our implementation uses Z3 to synthesize proof terms similar to
F invariants. The context will make the distinction clear. Modules in F
are translated to modules in DCIL, and we use visibility qualifiers to model information-hiding in DCIL—we omit this from our presentation here although this is included in our technical report.

DCIL distinguishes two types of classes. All (non-primitive)
types in F are translated to abstract classes \( T \). F values \( v:t \) are translated to instances of data classes \( D \), where \( D \) extends \( T \), the class corresponding to \( T \). Classes can be parametrized by a list of type parameters \( \alpha=\hat{\lambda} \) and also by a list of value parameters \( \hat{\lambda} \). Both kinds of classes include field and method declarations, although bodies of method declarations in T-classes are empty. Data classes include value constraints \( \hat{\lambda} \), which are analogous to
F’s pattern matching assumptions—we discuss this 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, \( (v \text{ isinst } D (\vec{\tau}, \vec{\alpha})) \) then \( e_1 \) else \( e_j \). 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 \( T (\vec{\tau}, \vec{\alpha}) \). Affine types are written \( !r \), as in FINE. DCIL includes a restricted form of type-level function \( \backslash \vec{x}.\tau_1 \tau_2 \) to represent dependent types. Type-level function application is denoted \( \tau \nu \). Kinds include \( \ast \) and \( A \) to categorize normal and affine types, respectively, and \( \tau \to \kappa \), the kind of type-level functions.

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. Modules in F are translated to modules in DCIL, and we use visibility qualifiers to model information-hiding in DCIL—we omit this from our presentation here although this is included in our technical report.

DCIL distinguishes two types of classes. All (non-primitive)
Figure 7. Static semantics of DCIL (selected rules)

4.2 Overview of DCIL

DCIL makes 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 $\star \rightarrow \kappa$ or $\kappa \rightarrow \kappa$, 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 rich 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 $T(\bar{\tau}, \bar{v})$) from data classes ($D(\bar{\tau}, \bar{v})$). 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. The body of a class $D$ with a value constraint $x \equiv v$, is checked with the assumption that its field $x$ holds values equal to $v$. When constructing an instance of such a class $D$, we check that the values provided for these fields satisfy the constraints.

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 $\Sigma$ that collect declarations of both $D$- and $T$-classes; $\Gamma$, a local typing environment; and $X$ a context containing usable affine assumptions.

The (WF-Decl) rule defines well-formedness of a data class declaration. The important aspect of this rule is that field declarations and method bodies are checked in a context $\Gamma$ that, in addition to bindings for the type and value parameters of the class, includes the value constraints, $\forall \bar{v}$, of the class declaration.

The (T-New) rule shows how the value constraints are accumulated when checking the branches of DCIL’s type-test construct. The set of affine assumptions $X$, $X'$ are split between the value $y$ being scrutinized and the branches. In the second premise, we compute value constraints $\forall \bar{v}$ required to unify the type of the pattern class $D(\bar{\tau}, \bar{v})$ with type $\tau_j$ of $y$. We check the true branch $e_2$ with an additional value constraint recording the result of successful type-test. Notice also that $e_j$ is checked in a context that includes variable bindings for each of the value parameters $\bar{x}$ for $D(\bar{\tau}, \bar{x})$. Our isinst construct is a macro that expands to multiple ct. instructions, where in the then-branch these instructions include projections of each of the fields corresponding to the value parameters of $y$. DCIL provides no other way to project fields corresponding to value parameters. The (T-New) rule requires that all the value constraints $\forall \bar{v}$ in a class are satisfied (the second premise) when the its constructor is called. Finally, the (T-App) rule shows the rule for method calls, and captures both type and term application in FINE. The salient feature here is the substitution of the actual parameter $v'$ for the formal name $x$ in the conclusion.

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)—notice that in the second premise the value $v$ is typified without any affine assumptions $X$. A similar restriction appears in the final premise of (TK-T)—value parameters of types are always of kind $\star$ and never use affine assumptions.

Finally, we show 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. The rule (TE-Beta) equates types related by $\beta$-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 $v_1 \equiv v_2$ 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. Values with affine types are held in a mutable store \( M \), where reads and writes to the store are destructive. In addition to showing that well-typed programs never get stuck, Theorem 2 guarantees that DCIL programs destruct affine values at most once.

**Theorem 2** (Soundness of DCIL).

1. \( VS, M, X, e, \tau, M', e' \) \( \vdash \text{wf}(S, \Gamma(M), X) \wedge S; \Gamma(M); X \vdash e : \tau \wedge (M, e) \rightarrow (M', e') \Rightarrow S; \Gamma(M'); X \vdash e' : \tau \) where \( X' = X \cup (\text{dom}(M') \setminus \text{dom}(M)) \) if \( \text{dom}(M') \geq \text{dom}(M) \)

2. \( I \vdash S, M, X, e, \tau, \text{wf}(S, \Gamma(M), X) \rightarrow S; \Gamma(M); X \vdash e : \tau \Rightarrow \exists e'. v = v' \lor \exists e', M'. (M, e) \rightarrow (M', e') \)

### 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 principal 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 illustrated 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 and value of proofs, \( pf : \ast \rightarrow \ast \), is represented in DCIL as an abstract class with a single type parameter: \( \text{class pf} < \ast : \ast > : \ast \). Dependent type constructors like \( \text{Eq} : \text{att} \rightarrow \ast \) are translated to abstract classes with value parameters: \( \text{class Eq:att} < x : \ast, y : \ast : > : \ast \). For instance, 

**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 \( \text{And elim} : \text{pf} < \text{And} \alpha \beta > \rightarrow \text{pf} < \alpha > \) data constructor is translated to the class \( \text{And elim} : \text{pf} < \text{And} \alpha \beta > \rightarrow \text{pf} < \alpha > \). The value parameter of \( \text{And elim} \) corresponds to a field that holds a \( \text{pf} < \text{And} \alpha \beta > \) value, but notice that this value parameter does not appear in the type \( \text{pf} < \alpha > \) constructed by \( \text{And elim} \). This is in contrast to the data constructors of dependent types. For example, the reflexivity axiom \( \text{Ref eq att} : \text{att} \rightarrow \text{pf} < \text{Eq att} < a : a > \) is translated to a class \( \text{data Eq att} : \text{att} \rightarrow \text{pf} < \text{Eq att} < a : a > > : \ast \). The value parameter of \( \text{Ref eq att} \) corresponds both to a single field declaration in the body of the class and additionally appears as an index in the type \( \text{pf} < \text{Eq att} < a : a > > \) that it constructs.

#### Translation of function types.

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

\[
\text{DepArrow}(\alpha_1 : \ast, \alpha_2 : \ast_1 : \rightarrow \ast) \rightarrow \ast \rightarrow \ast \\
\text{App}(x : \alpha_1) \\
\]

Class \( \text{DepArrow} \) takes two type parameters: \( \alpha_1 \) for the argument type and \( \alpha_2 \) for a type function—the return type of App is the result of applying \( \alpha_2 \) to the argument \( x \). Source-level types such as \( p : \text{print} \rightarrow \text{cred} \rho \) are translated to instances of DepArrow; in this case, \( \text{class(PCredP)} \) is the form of the \( \langle x : \text{print}(\text{cred}(x)) \rangle \) App \( \langle p : \text{print} \rangle \) (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 \( \text{cred} < p > \), analogous to the type returned by the source-level function. Each function type in FINE is translated to a distinct class (like \( \text{PcredP} \) in DCIL and overrides the App method suitably). A closure conversion step collects the free variables of a function and adds these as additional 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.

\[
\text{Arrow}(\alpha_1 : \ast, \alpha_2 : \ast_1 : \rightarrow \ast) \rightarrow \ast \rightarrow \ast \\
\text{App}(x : \alpha_1) \\
\]

**Translation of dependent pairs.** Dependent pairs are translated similar 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.

\[
\text{DepPair_A}(\alpha_1 : \ast, \alpha_2 : \ast_1 : \rightarrow \ast) \rightarrow \ast \\
\text{App}(x : \alpha_1, \alpha_2) \\
\]

The data class \( DA \) can be instantiated appropriately to represent specific source values. For example, the source value \( (s, \text{tok}) \) of type \( (\text{stat} * \text{States} s) \) is translated to a new DA value using the constructor application \( \text{DA} < s, \text{stat} \{ s, \text{tok} \} > \) and is given the type \( \text{DepPair_A} < s, \text{stat} \{ s, \text{stat} \} > \). 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.

#### Translation of a proof term.

To illustrate the use of value constraints, we restate a fragment of the source-level proof term discussed in §3.3, and reproduced below:

\[
\text{M}(s : \text{stat} \rightarrow \text{pf} < \text{imp} (\text{ln (Role me Plugin(s)}) \rightarrow \text{Derivable}(p s)>) > \\
\text{MP} (\text{MP} (\text{Derivable}(p s), \text{Derivable}(p s))) > \\
\]

This lambda-expression is translated to a class \( D \), a subclass of:

\[
\text{Arrow} < \text{DepArrow} < s : \text{stat} \{ \ldots \} > > , \text{pf} < \text{Derivable}(p s) > > \\
\]

Closure conversion adds the free variables \( s, p, a \) as value parameters of \( D \), to be instantiated when \( D \) is constructed. The body of the lambda-term is translated to the body of the overridden App method of \( D \). Recall that to give the sub-term \( \text{Ref lift bool} \) the type \( \text{pf} < \text{Eq bool} b \rightarrow \text{true} > \), the source checker appealed to the assumption \( b \equiv \text{true} \) in the context. However, separate compilation in DCIL requires the class \( D \) to be type checked independently of the context in which it is used. Value constraints in DCIL record source-level match assumptions for classes like \( D \). In this case, \( D \) includes a value constraint \( b \equiv \text{true} \), and at each constructor application of \( D \) we check that the constraint is satisfied.

**Theorem 3** below states that the translation preserves types—\( \| \| \) denotes the translation of environments, and the judgments that use \( \rightarrow \) stand for translation of types and terms. Informally, Theorem 3 states that for every source program \( e \) derefined to \( e' \) and given the source type \( \tau, e' \) is translated to a target expression \( \hat{e} \) together with class declarations \( \Sigma \). In an environment that collects all the translated class declarations, \( \hat{e} \) is given the target type \( \tau \), the translation of the source type \( \tau \). Although the translation of types itself produces additional class declarations, these are always contained in the set \( \Sigma \) of declarations produced by the translation of expressions.
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 a parser and the binary writing libraries of the F# compiler. Our application programs, for the most part, are reference monitors—security-critical kernels of applications that are expected to be compact and highly reliable. 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 be an increase in the size of binaries by more than an order of magnitude. Although binaries are large, the time required to typecheck them remains quite low. We also report on an experiment with our largest benchmark, where through the use of a specialized custom solver, we were able to improve code size by more than 25X. In summary, our results indicate that end-to-end security verification is possible for many kinds of real-world programs, and, with improvements in certifying solvers, the overhead of carrying proofs can be small.

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 check target programs (TC); the size in bytes of .NET assemblies that do not include proof terms (NoPf); and the size of assemblies that do include proofs (Pl). 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. Automation is more interesting—it implements policy to enforce 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 to 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 I m, where I 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 contains a main event loop serving requests from an F# application. As server-side program, we expect that the proof terms produced for the verification of HealthWeb could be logged at runtime to construct audit trails of authorization decisions.

ConfRM is a reference monitor based on Confidence, a widely used conference management tool (Krishnamurthi 2003). This application was previously implemented and described in detail by Swamy et al. (2009) 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 leaking the contents of emails from one principal to another.

Second, we provide a way for plugins to selectively share information with each other via the cookie store. Permissions in Lookout are granted by one principal to another, e.g., the permission Permits p q a records a permission granted by p to q to perform the action a. This feature of Lookout provides a framework on which to build secure plugin mashups. When storing a value in the cookie store, a plugin can register an authorization function that mediates access to that cookie. Using this facility, the cookie value returned by the getCook function is given the type \( \langle \text{string} \mid \text{Derivable (Permits owner p (ReadCookie name e))} \rangle \), indicating that the reference monitor returns a cookie value to the principal p only when authorized by the cookie’s owner.

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 the time taken to type check DCIL programs is low. For example, typechecking the 51MB ProofLib takes less than 7 seconds, comparable to the time spent by pexify, the standard .NET bytecode verifier on assemblies of a similar size. 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.

In the remainder of this section, we examine why Z3 proofs are as big as they are, and report on an experiment that indicates that 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 25x 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—we do not claim that our simple solver is a replacement of 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

Our approach of compiling FINE to DCIL is an instance of proof-carrying code (PCC) (Neucla 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 non-interference property. Also related is Barthe et al. (2007) type system for non-interference 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 enforce- ment 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. Prior work by Swamy et al. (2009) 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 put our compiler to use in the construction and verification of the security critical modules of a number of applications. Although verification times for DCIL are already relatively low, 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.

References

C. Flanagan, S. N. Freund, and A. Tomb. Hybrid types, invariants, and refinements for imperative objects. In F00L/WOOD ’06, 2006.


